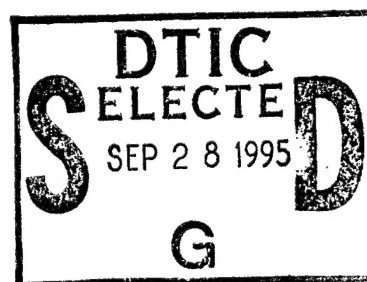


A TRIDENT SCHOLAR PROJECT REPORT

NO. 227

“Optimization of Linearly Constrained Indefinite Functions”



19950927 121

UNITED STATES NAVAL ACADEMY
ANNAPOLIS, MARYLAND

This document has been approved for public
release and sale; its distribution is unlimited.

DTIC QUALITY INSPECTED 5

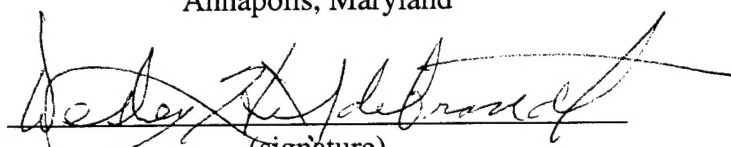
REPORT DOCUMENTATION PAGE			Form Approved OMB no. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour of response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspects of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE 9 May 1995		3. REPORT TYPE AND DATES COVERED
4. TITLE AND SUBTITLE Optimization of linearly constrained indefinite functions			5. FUNDING NUMBERS	
6. AUTHOR(S) Wesley A. Hildebrandt				
7. PERFORMING ORGANIZATIONS NAME(S) AND ADDRESS(ES) U.S. Naval Academy, Annapolis, MD			8. PERFORMING ORGANIZATION REPORT NUMBER USNA Trident report; no. 227 (1995)	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES Accepted by the U.S. Trident Scholar Committee				
12a. DISTRIBUTION/AVAILABILITY STATEMENT This document has been approved for public release; its distribution is UNLIMITED.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Global optimization is the process of finding a best solution among many possible solutions to a problem involving the minimization or maximization of some desired "cost" function. In general, many problems arising from practical applications can be formulated using both an objective function to be optimized (cost, profit, etc.) and a set of restrictions on the allowed solutions. In some cases this objective function may be linear, in which case the problem may yield to linear programming techniques. In other cases it may be entirely concave or convex. In these cases the solution may again be easy to obtain since certain properties of the function allow special searching techniques to locate the optimum solution. In the hardest cases the objective function is indefinite, which means that it can have many local minima, none of which satisfy any special properties. Furthermore, these problems are usually bounded by constraints, which restrict the allowed values of the individual variables. In the majority of real problems the constraints will be linear, in which case the optimization problem can be approached using matrix algebra techniques. This paper will present two methods for optimizing indefinite functions with linear constraints, and computational results obtained using each method.				
14. SUBJECT TERMS global optimization; nonlinear programming; stochastic methods; branch-and-bound methods; parallel processing; indefinite functions.			15. NUMBER OF PAGES	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNCLASSIFIED	

“Optimization of Linearly Constrained Indefinite Functions”

by

Midshipman Wesley A. Hildebrandt, Class of 1995

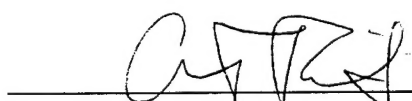
United States Naval Academy
Annapolis, Maryland


(signature)

Certification of Advisor Approval

Associate Professor Andrew T. Phillips

Computer Science Department


(signature)

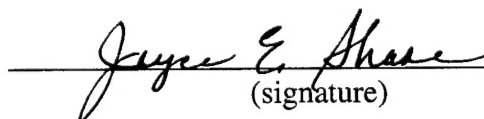
5/9/95
(date)

Accession For	
NTIS	CRA&I <input checked="" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification _____	
By _____	
Distribution / _____	
Availability Codes	
Dist	Avail and / or Special
A-1	

Acceptance for the Trident Scholar Committee

Professor Joyce E. Shade

Chair, Trident Scholar Committee


(signature)

9 May 95
(date)

Abstract

Global optimization is the process of finding a *best* solution among many possible solutions to a problem involving the minimization or maximization of some desired “cost” function. In general, many problems arising from practical applications can be formulated using both an objective function to be optimized (cost, profit, etc.) and a set of restrictions on the allowed solutions. In some cases this objective function may be linear, in which case the problem may yield to linear programming techniques. In other cases it may be entirely concave or convex. In these cases the solution may again be easy to obtain since certain properties of the function allow special searching techniques to locate the optimum solution. In the hardest cases the objective function is indefinite, which means that it can have many local minima, none of which satisfy any special properties. Furthermore, these problems are usually bounded by constraints, which restrict the allowed values of the individual variables. In the majority of real problems the constraints will be linear, in which case the optimization problem can be approached using matrix algebra techniques. This paper will present two methods for optimizing indefinite functions with linear constraints, and computational results obtained using each method.

Keywords: global optimization, nonlinear programming, stochastic methods, branch-and-bound methods, parallel processing, indefinite functions.

Preface

This paper considers two methods for finding the global optimum of general indefinite functions bounded by linear constraints. Chapter I provides an introduction to the art and science of global optimization, and then Chapters II and III discuss two methods, a stochastic algorithm and a branch-and-bound algorithm, which are applied to this problem. Chapters IV and V describe the two computational systems on which these methods are implemented, the Parallel Virtual Machine system and the MasPar MP-1 supercomputer, and their relative advantages and drawbacks. Chapters VI and VII contain the test results obtained from these two methods, and Chapter VIII discusses some general results and areas for further work. Finally, Appendix A includes example output from a sample problem run using each method, and Appendix B details a new theoretical development for the second method.

Acknowledgements

I would like to offer my appreciation to all those without whose help, support, and encouragement this project would not have been possible. Laura, thank you for understanding and for supporting me throughout this project. Mike, thanks for keeping the rest of my life in perspective. Cathie, your countless hours spent fixing problems you didn't create are not forgotten. LCDR Traub, thank you for allowing me to prioritize this project over other important in-company functions. I would also like to thank everyone in the Computer Science Department of the United States Naval Academy for their support; and of course, my deepest gratitude must go out to my advisor, Dr. Andrew T. Phillips. I would have been lost without his vital guidance and instruction.

Contents

Abstract.....	1
Preface.....	2
Acknowledgements	3
Contents	4
Figures.....	6
Tables	7
 I. Introduction.....	 8
1. Naval Berthing	8
2. Business Applications.....	9
3. VLSI Chip Design.....	10
4. Other Examples.....	11
5. Global Optimization	12
6. Indefinite Functions	12
7. Constraints	15
8. Overview.....	16
 II. Stochastic Method.....	 18
1. Initial Starting Points	18
2. Gradient Projection.....	20
3. Leaving Constraints	24
4. Step Length.....	25
5. Optimal Bayesian Estimate.....	26
6. Termination.....	29
 III. Guaranteed Bound Method	 31
1. Objective Function Limitations	31
2. Overview of Algorithm.....	32
3. Subregion Elimination	37
4. Updating the Bounds.....	40
5. Divide and Conquer	41
 IV. PVM Network	 45
1. Description of the PVM Package	45

2. PVM and the Stochastic Method	47
3. PVM and the Guaranteed Bound Method.....	49
V. MasPar MP-1.....	51
1. Stochastic Method.....	51
2. Guaranteed Bound Method.....	54
3. Maximum Possible Speedup.....	55
VI. Stochastic Method Results	57
1. Stochastic Method Implementation	57
2. "Random" Test Problems	58
3. Global Minima	58
4. Largest Problems Solved	61
5. Local Minima	61
6. Eigenvalues and Problem Difficulty.....	62
7. Time of Execution.....	64
8. Number of Trials.....	64
9. Time per Trial	66
10. Concluding Remarks.....	67
VII. Guaranteed Bound Method Results	68
1. Sequential Execution Time.....	68
2. Constraint Dependence.....	69
3. Parallel Execution Time	71
4. Concluding Remarks.....	72
VIII. Conclusions	73
1. General Notes	73
2. Future Work	73
Appendix A: Sample Program Output	75
1. Stochastic Method.....	75
2. Guaranteed Bound Method.....	76
Appendix B: Subregion Elimination	79
References	81

Figures

1.1	Simple Concave Function	13
2.1	Comparison of Stopping Rules	29
2.2	Stopping Rule Comparison (large ω)	30
3.1	Indefinite Function Example	33
3.2	Depth of Constrained Function	34
3.3	Initial Convex Underestimator	36
3.4	Example of Subregion Elimination	39
5.1	Maximum Speedup for 10% Sequential Code.	56
6.1	Observations of the Global Minimum	60
6.2	Number of Observed Local Minima	62
6.3	Magnitude of the Concave Eigenvalues	63
6.4	Setup Time and Run Time.	65
6.5	Time per Trial for Different Size Problems.	66
7.1	Sequential Run Time vs. Problem Size	69
7.2	Effect of Constraints on Running Time.	70
7.3	Execution Time vs. Problem Size	71

Tables

4.1	Setup Time for Constant Number of Constraints	48
4.2	Setup Time for Constant Number of Variables	49
5.1	Comparison of MasPar MP-1 and PVM Speed	53
6.1	Global Solution and the Stochastic Method	59
6.2	Statistics for Three Largest Problems	61
6.3	Number of Trials Required	65

I. Introduction

Global optimization is the process of finding a *best* solution among many possible solutions to a problem involving the minimization or maximization of some desired “cost” function. Developing methods to quickly identify the global optimum for a given situation is an important area of research, since many practical applications can be modeled as optimization problems. To motivate this research, this paper begins with several case studies in which optimization techniques are used.

1. Naval Berthing

The assignment of naval vessels to specific berths at a shipyard has always been a difficult and costly problem. Certain essential services are available only at some subset of the berths, and maintenance is often only available or is least expensive at certain others. In addition, the draft of the ships and other size restrictions preclude them from mooring at berths of insufficient size. Planned deployment dates and even the personal wishes of the military chain of command can also affect the set of possible assignments. With all of these competing and conflicting restrictions, the process of making *any* assignment, let alone an “optimal” one, is quite difficult.

The current method at nearly all shipyards is for people to get together once a week for several hours and create a detailed plan which assigns each ship to a specific berth on a specific pier. Since this means that planning is rarely done for more than a week in advance, many ships are subject to what is commonly referred to as the “Monday

shuffle,” during which various ships are forced to physically relocate, a process which can claim all the resources of those ships involved for up to half a day every week. This is clearly not an efficient way to operate a shipyard and wastes many tax dollars that could be put to better use elsewhere.

An optimization approach to this problem is to first model the cost, in terms of time and money, of moving these ships as a function of all the factors mentioned above. This can be a very complex model, requiring in some cases ten thousand or more variables. In the specific instance of the submarine base in San Diego, the optimization algorithm (based on linear programming) which has been implemented for this process is able to achieve a solution within 6% of the optimal solution in thirty minutes [4]. This example shows that optimization techniques are currently useful, and they will continue to be used even more as the capabilities of the algorithms increase.

2. Business Applications

Typical business situations also often involve optimization. For example, an office equipment company might produce copiers, fax machines, and phone equipment. Each type of equipment requires a different set of resources for its production, but there may be some overlap between these resources, implying that not all production operations can be performed at the same time. Because of this overlap, the company can not operate at maximum production capacity for all three types of equipment simultaneously. In addition, each type of equipment will have a production cost affected

by various factors, including the speed at which it is produced. They will also each have a profit potential based on their production cost and their estimated selling value.

In this case the clear goal is to maximize the company's profit. To do this using optimization algorithms, the profit is first formulated as a mathematical objective function of the input variables, which in this case are the various factors affecting the cost of production. The function is then optimized, and the value of those variables at the global solution will guide the company in allocating its resources for the production of the three types of equipment so as to maximize its profit.

3. VLSI Chip Design

Another well known example involving the use of optimization is Very Large Scale Integrated (VLSI) chip design. VLSI circuits are characterized by a very large number of transistors on a single chip, typically in the hundreds of thousands or millions, arranged into subcomponents that must be linked together in specified ways to produce the overall functionality of the chip. There are two primary goals when fabricating a specific chip.

One goal is the speed response of the chip. Chip components that communicate most frequently should ideally be placed closer together, so that the performance of the chip (in terms of speed) will be increased. Additionally, the size of the final design should be minimized. The cost of cutting a VLSI chip from the original silicon wafer is directly related to the number of chips that will fit on a single wafer, and therefore the size of the chip. In optimizing this process the first step is to construct an objective function that

mathematically represents the size of the chip and also the placement of the individual components on that chip. Then this objective function is minimized, at which point the solution shows the design technicians where to place the components for maximum performance while also reducing the surface area of the chip and the cost of fabricating it.

4. Other Examples

Many other engineering problems can be modelled as optimization problems. One example is the design of support trusses, in which the objective function is designed to maximize the strength and stiffness of the truss, by taking into account which beams are included in the truss, where those beams are placed, and the size of those beams. The constraints on this model are the requirements that specify how large a load the truss must support, and where the load will be placed [2].

Another engineering example is a model used to optimize the purification of underground water supplies by removing pollution. In this case the model is designed to minimize pollution, and takes as variables the locations and rates of pumping out the dirty water and pumping in the clean water. The model can also be designed with the level of pollution as a constraint, in which case the objective function is written to minimize either the time or the cost of getting the pollution reduced to an acceptable level [9].

Finally, in the world of business and finance, it is often desirable to generate the maximum return from an investment portfolio with minimum risk. An objective function is written which models the possible choices for the types of investments, or even the

individual stocks and other investment tools, and is weighted depending on how much risk is acceptable, and how high a return is desirable [11].

5. Global Optimization

In all of these examples the key to getting the best results is global optimization. A good base operations officer, a savvy business manager, or a skilled chip design technician could all come up with “locally” optimal solutions to their individual problems. Their hand-crafted solutions, although better than many other possible solutions, are usually not the best available. By modelling the processes mathematically and using optimization algorithms to solve them, a much better solution is typically obtained. In addition, these solutions can be found automatically, utilizing fewer resources than the equivalent human requirements. For this reason the ultimate goal of all optimization algorithms is to arrive at a solution that is as close to the optimal solution as is possible within some given time frame.

6. Indefinite Functions

Once a process has been identified as a good candidate for optimization, it must first be modelled by a mathematical function. In optimization this function is known as the objective function. This function can always be written so that the goal is to minimize its value. Although some objectives, such as profit, should be maximized, it is always possible to write the function in this form by negating it if it should be maximized; that is, $\max f(x)$ is equivalent to $-\min -f(x)$. In some of the most simple cases, this function could

be linear, which is to say that it is a linear combination of the input variables. Problems of this nature admit themselves well to linear programming techniques. Other problems, while not linear, may have certain properties that allow them to be approached using special algorithms from network flow and graph theory. The optimization algorithm that is currently in use at the submarine base in San Diego has been modelled as a linear objective function which represents the availability of facilities and the goal of minimizing ship relocations.

On a more complex level there are functions which are either concave or convex. A concave function is one which looks like an inverted "bowl," with a single maximum point and contours which decrease in all directions. Figure 1.1 shows an example of a

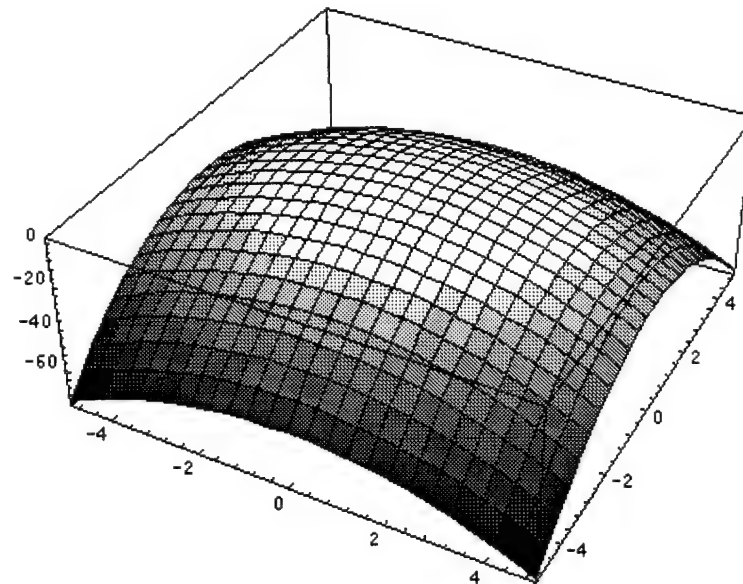


Figure 1.1: Simple Concave Function

simple concave function, $\phi(\mathbf{x}) = -x_1^2 - 2x_2^2$, plotted over the region $-4 \leq x_1, x_2 \leq 4$, with a single unique maximum at (0,0). Convex functions are just the opposite. They

have a single point of minimum function value, and the function value increases in all directions from that point. It is important to note that the “curvature” of these functions need not be identical in all dimensions. That is, a convex function may have elliptical, rather than circular, horizontal cross-sections. These graphical illustrations are only applicable to functions in two dimensions; in higher dimensions it is difficult or impossible to picture concavity and convexity. Therefore, the mathematical definitions of these terms must be described. Given a function $\phi: \mathcal{R}^P \rightarrow \mathcal{R}^Q$, the function is concave if for any two points $\mathbf{a}, \mathbf{b} \in \mathcal{R}^P$ in its domain, the affine approximation to the function value which interpolates these two points also underestimates the function ϕ at every point on the line between them. This line can be represented as $L = \{\mathbf{a} + t(\mathbf{b} - \mathbf{a}) : t \in (0,1)\}$, and then the affine approximation will be $\bar{\phi}(t) = t \cdot \left(\frac{\phi(\mathbf{b}) - \phi(\mathbf{a})}{\mathbf{b} - \mathbf{a}} \right) + \phi(\mathbf{a})$. If the function is convex, then this affine approximation will *overestimate* the function ϕ at every point on this line. The minima for convex functions are very easy to find, since each function only has one! On the other hand, concave functions can have many minima, but because these minima must be at vertices of the feasible region [19], special algorithms can be designed to make use of this knowledge to decrease the search time.

The hardest class of optimization problems to solve are those with indefinite objective functions. An indefinite function is one which is neither strictly concave nor strictly convex, but has many regions containing local minima and maxima throughout its domain. Since in general the minima for these functions can be anywhere in the feasible region, it is very difficult to design an algorithm which will solve all types of these

problems. Current algorithms rely on special characteristics of the functions to solve them; in some cases the objective function may be required to be quadratic, while in others the function may have to be separable. Also, in virtually all cases the function is required to have continuous derivatives. A major goal in nonlinear optimization is to develop algorithms which will work on the greatest possible range of functions.

Both the VLSI chip design and the business production problems fall into this hardest category of problems. Since many real life scenarios are modelled by indefinite functions, this area of optimization is very important, and will become more so as the algorithms available become both more effective and more general.

7. Constraints

In all of the examples given, and in fact in most optimization problems, constraints are put on the input variables. In the case of a business application, one constraint could be that a certain piece of production equipment can only be producing one product at a time. Another may be that the number of hours the equipment can be operating each day is limited. Other constraints can involve multiple variables. For example, if one component of a chip is placed in a certain position, no other component may be placed so as to overlap with it. In each of these cases, the constraints can be mathematically formulated as linear equalities or inequalities. Each of these constraints serves to "bound" the set of allowed solutions to the problem of interest, by enforcing practical and necessary restrictions on the solution set.

8. Overview

Based on the variety of practical applications which can be modeled as global optimization problems with linear constraints, this paper examines the general problem of finding the

$$\begin{array}{ll} \text{global min} & \phi(\mathbf{x}) \\ \text{subject to} & \mathbf{Ax} \leq \mathbf{b} \end{array}$$

where $\phi(\mathbf{x})$ can be any continuously differentiable, yet indefinite, function. The feasible space for this objective function is determined by the set of linear constraints given by $\mathbf{Ax} \leq \mathbf{b}$.

This paper presents two algorithms for effectively solving this global optimization problem. Chapter II discusses a stochastic method, which is a probabilistic approach applicable to virtually all functions with continuous derivatives. The guaranteed bound method, which requires that the objective function be separable and strictly concave or convex in each dimension, is then presented in Chapter III.

Following the discussion of the two algorithms, Chapters IV and V will present the means and machines used to implement them. In particular, these chapters will describe the Parallel Virtual Machine (PVM) network, which is a heterogeneous collection of workstations that appears to be a single virtual machine, and also the MasPar MP-1, which is a single-instruction, multiple-data (SIMD) massively parallel supercomputer that relies on great numbers of relatively weak processors to accomplish difficult tasks quickly.

The next two chapters of this paper will present the test results from the application of these algorithms to both known and randomly generated test problems. They will be followed by Chapter VIII, which will discuss general conclusions and areas for further work. Following the main text is Appendix A, with sample output from each of the methods, and Appendix B, which details a theoretical development on the efficiency of subregion elimination for the guaranteed bound method. This paper closes with the list of references.

II. Stochastic Method

The first algorithm studied in this project is a stochastic method which uses repeated local minimization in an attempt to determine *all* local minima of the objective function within the constraint region. For each trial an initial starting point in the feasible region is chosen randomly, and a local minimization method based on the work of Rosen ([20], [21]) is used to find a local minimum from that starting point. Since it is not possible in general to deterministically know precisely how many minima there are in the feasible region, and therefore when to stop searching for new minima, this algorithm uses an optimal Bayesian estimate based on the number of distinct local minima observed, and terminates when the number of trials executed achieves that estimate.

1. Initial Starting Points

The first step in this algorithm is to generate initial starting points. In order for the Bayesian estimate to be valid, these initial starting points must randomly and uniformly cover the entire feasible region defined by the constraints. To make this process easier, a hyperrectangle of minimum size is constructed which contains the entire feasible region. This hyperrectangle is defined as

$$H = \{x_i: l_i \leq x_i \leq u_i, i = 1, 2, \dots, n\},$$

where \mathbf{l} and \mathbf{u} are n -dimensional vectors representing the lower and upper bounds l_i and u_i for $i = 1, 2, \dots, n$, respectively, on the problem variables \mathbf{x} . One obvious method of

producing random starting points is to generate them uniformly throughout the hyperrectangle, and throw out any that are not actually in the feasible region. To determine if a point is inside the feasible region, it is only necessary to check it against the constraints. The problem statement defines the constraint requirements as $A\mathbf{x} \leq \mathbf{b}$, where A is an $m \times n$ matrix (m constraints by n variables), $\mathbf{x} \in \mathcal{R}^n$ is a point in n -space, and $\mathbf{b} \in \mathcal{R}^m$ is a vector in m -space. To check if any given point is feasible, it is only necessary to check the condition $\mathbf{b} - A\mathbf{x} \geq \mathbf{0}$.

However, empirical evidence (based on randomly generated problems) has shown that as the number of problem variables increases, the number of random points produced which are interior to the feasible region becomes exponentially small. Also the overall execution time tends to be dominated by this "point generation" step. Hence, an alternative method for picking these random points is required.

One such method is to pick a random point in the hyperrectangle, and if it is not already in the feasible region, then it may be projected into the region. In particular, this algorithm first determines two points \mathbf{s} and \mathbf{r} on the boundary of the feasible region and located sufficiently far apart. Since all of the random problems used with this method are constructed so that the origin is feasible, \mathbf{s} may be set to $(0, 0, \dots, 0)$. The point \mathbf{r} is the solution to the linear program

$$\begin{aligned} \max \quad & \sum x_i \\ \text{s.t.} \quad & A\mathbf{x} \leq \mathbf{b}, \end{aligned}$$

and may be thought of as the point of the feasible region which is "farthest" from \mathbf{s} (the origin) in the direction $(1, 1, \dots, 1)$. Hence, if a point is generated that is outside of the

region, it is projected toward a randomly chosen point on the chord connecting \mathbf{s} and \mathbf{r} . Since these two points are in the feasible region, and the linearly constrained region is a convex polyhedron, all points on the line between them must also be in the region. To ensure that the starting points are uniformly random throughout the region, the algorithm does not project this random point exactly onto the line, but instead only projects it some random distance toward the line, stopping at some point inside the feasible space. Empirical data using this method for choosing starting points shows a substantial improvement in the amount of program execution time that is now spent finding starting points.

In summary, the algorithm used to generate these points is as follows:

1. Pick a uniformly distributed random point \mathbf{x} in the hyperrectangle.
2. If \mathbf{x} is in the feasible region then return it.
3. If not,
 - a. Set $\mathbf{t} = [\mathbf{s} + \text{rnd}() \cdot (\mathbf{r} - \mathbf{s})] - \mathbf{x}$, where $\text{rnd}()$ returns a uniformly random number in $(0, 1)$.
 - b. Set $\alpha_1 = \min_{\text{s.t. } (A(\mathbf{x} + \alpha\mathbf{t}) \leq \mathbf{b})} \alpha$
 - c. Set $\alpha_2 = \max_{\text{s.t. } (A(\mathbf{x} + \alpha\mathbf{t}) \leq \mathbf{b})} \alpha$
 - d. Set $\alpha = \alpha_1 + \text{rnd}() \cdot (\alpha_2 - \alpha_1)$
 - e. Return $\mathbf{x} + \alpha\mathbf{t}$ as the starting point.

2. Gradient Projection

After finding an initial feasible point, a method is required to move from that point to a local minimum. In an unconstrained minimization problem, one simple approach is that of steepest descent. In this approach the direction of the negative

gradient is followed until a local minimum is found. However, when the problem is constrained, the algorithm can not simply follow the negative gradient, as that direction may conflict with one or more of the constraints. As a result, if the negative gradient is projected onto the set of “active” constraints¹, this resulting direction can again be followed to a local minimum. This method is called the gradient projection method [20].

The general goal of all basic descent methods for finding a local minimum is to move in a direction that results in a lower function value until no more such movement is possible, at which point a local minimum has been found. If $\mathbf{d} \in \mathcal{R}^n$ is a normalized “direction” vector ($\|\mathbf{d}\|_2 = 1$) representing the intended descent direction, and α represents the step length to be taken in the direction of \mathbf{d} , then the current iterate \mathbf{x} can be used to find the next iterate \mathbf{x}' using the relationship $\mathbf{x}' = \mathbf{x} + \alpha\mathbf{d}$. Furthermore, the Taylor series expansion of $\phi(\mathbf{x}')$ about the point \mathbf{x} is then given by

$$\phi(\mathbf{x}') = \phi(\mathbf{x} + \alpha\mathbf{d}) = \phi(\mathbf{x}) + \nabla\phi(\mathbf{x})^T \alpha\mathbf{d} + \frac{1}{2}\alpha^2 \mathbf{d}^T \nabla^2\phi(\mathbf{x})\mathbf{d} + \dots$$

In a small neighborhood of \mathbf{x} , that is when α is small, the higher order terms (those with α^2 and higher) become insignificant. So to enforce the requirement that $\phi(\mathbf{x}') < \phi(\mathbf{x})$ (that is, \mathbf{x}' is an improvement over \mathbf{x}), it is only necessary that $\nabla\phi(\mathbf{x})^T \mathbf{d} < 0$ at every step. This relation is called the “descent condition.” Note that if $\mathbf{d} = -\nabla\phi(\mathbf{x})$ is feasible, then this condition results in the maximum *local* decrease in $\phi(\mathbf{x})$.

Clearly, at any given point in the feasible region for this problem, some subset (possibly empty) of the constraints will be active. If A is written as

1. “Active” constraints are those for which the constraint relation $A\mathbf{x} \leq \mathbf{b}$ is strict equality.

$$\begin{bmatrix} \mathbf{a}_1 \\ \dots \\ \mathbf{a}_m \end{bmatrix} = \begin{bmatrix} a_{1_1} & \dots & a_{1_n} \\ \dots & & \\ a_{m_1} & \dots & a_{m_n} \end{bmatrix}$$

and \mathbf{b} is written as $[b_1 \dots b_m]$, then the active constraints are those \mathbf{a}_i for which $\mathbf{a}_i \cdot \mathbf{x} = b_i$. If there are c of these active constraints, then the corresponding "active constraint matrix" A_c will be defined as

$$A_c = \begin{bmatrix} \mathbf{a}_{c_1} \\ \dots \\ \mathbf{a}_{c_c} \end{bmatrix}.$$

The gradient projection method initially attempts to find a feasible direction which will not change the set of active constraints. To do so this direction must be chosen from the set of directions \mathbf{d} for which $\mathbf{a}_i \cdot \mathbf{d} = 0$ for all i in the current active set; that is, \mathbf{d} is orthogonal to the normal vector \mathbf{a}_i of *every* active constraint.

Those vectors \mathbf{d} for which $A_c \mathbf{d} = \mathbf{0}$ form a subspace S of the feasible region on which the next point will be found. This subspace will have dimension k , where $k = n - c$. Since \mathbf{d} is orthogonal to each row of A_c , the subspace T which is orthogonal to S can be defined as those vectors $\mathbf{v} = A_c^T \mathbf{u}$, where \mathbf{u} is any vector in \mathcal{R}^c . The subspace T is orthogonal to the subspace S since any $\mathbf{v} \in T$ and any $\mathbf{d} \in S$ satisfy

$$\mathbf{d}^T \mathbf{v} = \mathbf{d}^T A_c^T \mathbf{u} = (A_c \mathbf{d})^T \mathbf{u} = \mathbf{0}^T \mathbf{u} = 0.$$

Since every vector can be separated into components from each of these subspaces, the negative gradient can be written in two parts. The part of the gradient that lies in S is \mathbf{d} ,

the next search direction; this is, the “unconstrained” negative gradient projected onto the active constraints. If the gradient $\nabla\phi(\mathbf{x})$ is denoted \mathbf{g} , then

$$-\mathbf{g} = \mathbf{d} + A_c^T \mathbf{u}. \quad (2.1)$$

To calculate \mathbf{d} , the requirement that $A_c \mathbf{d} = \mathbf{0}$ may be used to compute \mathbf{u} . In particular,

$$-A_c \mathbf{g} = A_c \mathbf{d} + (A_c A_c^T) \mathbf{u}$$

so that

$$A_c \mathbf{d} = -A_c \mathbf{g} - (A_c A_c^T) \mathbf{u} = \mathbf{0}.$$

Solving for \mathbf{u} yields

$$\mathbf{u} = -\left(A_c A_c^T\right)^{-1} A_c \mathbf{g}.$$

This can be substituted in Equation 2.1 to arrive at a search direction \mathbf{d} in terms of only the current gradient \mathbf{g} and the set of active constraints A_c :

$$\begin{aligned} \mathbf{d} &= -\mathbf{g} + A_c^T \left(A_c A_c^T\right)^{-1} A_c \mathbf{g} = -\left[\mathbf{I} - A_c^T \left(A_c A_c^T\right)^{-1} A_c\right] \mathbf{g} = -\mathbf{P} \mathbf{g}, \\ \text{where } \mathbf{P} &= \left[\mathbf{I} - A_c^T \left(A_c A_c^T\right)^{-1} A_c\right]. \end{aligned}$$

This matrix \mathbf{P} , which is applied to the gradient, is known as the projection matrix for the subspace S under the current set of active constraints at the point \mathbf{x} . When applied to any vector in \mathcal{R}^n , \mathbf{P} will yield the “projection” of that vector onto the subspace S . Once this matrix has been applied to \mathbf{g} , there are two possibilities, depending on whether or not \mathbf{d} is nonzero. Assuming \mathbf{d} is nonzero, it is easy to show that the descent condition $\nabla\phi(\mathbf{x})^T \mathbf{d} < 0$ holds. By the original formula,

$$\mathbf{g} + \mathbf{d} = -A_c^T \mathbf{u},$$

so the vector $\mathbf{g} + \mathbf{d}$ is in the subspace T and must therefore be orthogonal to \mathbf{d} (which is in the subspace S). This means that $(\mathbf{g} + \mathbf{d})^T \mathbf{d} = \mathbf{0}$, so that

$$\nabla \phi(\mathbf{x})^T \mathbf{d} = \mathbf{g}^T \mathbf{d} = \mathbf{g} + \mathbf{d} - \mathbf{d}^T \mathbf{d} = \mathbf{g} + \mathbf{d}^T \mathbf{d} - \mathbf{d}^T \mathbf{d} = -\|\mathbf{d}\|_2^2 < 0$$

and therefore \mathbf{d} must be a valid descent direction restricted to the current set of active constraints.

3. Leaving Constraints

If the projected gradient \mathbf{d} turns out to be zero, then it is not possible to find a direction which maintains the current constraints and which will also result in a decrease in the function value $\phi(\mathbf{x})$. At this point then,

$$\mathbf{g} + A_c^T \mathbf{u} = \mathbf{0}. \quad (2.2)$$

This in fact is a necessary condition for the current point \mathbf{x} to be a minimum restricted to S , the subspace defined by the active constraints; however, this condition is not sufficient. If it is also true that every component of \mathbf{u} is nonnegative, then Equation 2.2 is both necessary and sufficient for \mathbf{x} to be a Karush-Kuhn-Tucker point¹ [10].

If, on the other hand, one or more of the components of the so-called Lagrange multipliers \mathbf{u} are negative, then the conditions for this point to be a KKT point (i.e., a minimum) on the subspace S of active constraints are not satisfied, and by removing one of the active constraints and searching in the resulting less restrictive subspace, a lower

1. A Karush-Kuhn-Tucker point is one at which the gradient, taken with respect to the applicable constraints, is zero. As in an unconstrained problem, this implies that the point is a maximum, minimum, or "saddle" point relative to the constraints.

function value will be found. To determine which active constraint to remove, the constraint corresponding to the most negative component of \mathbf{u} may be selected. If $A_{c'}$ is the updated version of A_c with this single active constraint removed, and if \mathbf{d}' represents the new direction and \mathbf{u}' is the new Lagrange multiplier vector in \mathcal{R}^{c-1} , then Equation 2.1 becomes

$$-\mathbf{g} = \mathbf{d}' + A_{c'}^T \mathbf{u}'.$$

This direction \mathbf{d}' will be the new descent direction.

4. Step Length

Once a valid direction of descent \mathbf{d} has been determined, a step length α must be computed. When moving away from \mathbf{x} in the direction of \mathbf{d} , in a small neighborhood of \mathbf{x} the value of $\phi(\mathbf{x} + \alpha\mathbf{d})$ must decrease and $\mathbf{x} + \alpha\mathbf{d}$ must remain in the feasible region. The optimal step length α^* is the one which gives the greatest decrease in $\phi(\mathbf{x})$ along the direction \mathbf{d} subject to the constraints. The first step in determining the optimal distance is to find the maximum α , denoted $\bar{\alpha}$, for which $\mathbf{x} + \alpha\mathbf{d}$ remains feasible; hence $\alpha = 0$ and $\alpha = \bar{\alpha}$ bracket the desired optimal value of $\alpha = \alpha^*$. Then the desired α^* is found which solves

$$\begin{aligned} \min \quad & \phi(\mathbf{x} + \alpha\mathbf{d}) \\ \text{s.t.} \quad & 0 \leq \alpha \leq \bar{\alpha}. \end{aligned}$$

If $\alpha^* = \bar{\alpha}$ then a new constraint must be added to the set of active constraints. In any case, the new point \mathbf{x}' will be defined by $\mathbf{x}' = \mathbf{x} + \alpha\mathbf{d}$, and the algorithm will continue from this new point until it finally reaches a local minimum.

5. Optimal Bayesian Estimate

Once the stochastic algorithm has found a local minimum, it must compare this point against all of the local minima it has found so far and decide if it new. If it is not a new minimum, then a new random starting point is generated and the algorithm begins again. On the other hand, if it *is* a new minimum, then the optimal Bayesian estimate must be checked to determine when enough unique minima have been found to provide a reasonable assurance of having actually found the *global* minimum. This estimate is based on two stopping rules which are derived from the work of Boender and Rinnooy Kan [3], and are restated in McLaughlin [12]. The first stopping rule is based on the following theorem:

Theorem 2.1: Let ω be the number of different observed local minima obtained as a result of performing N uniformly distributed random local searches. The optimal Bayesian estimate of the number of local minima, for $N \geq \omega + 3$, is:

$$\frac{\omega \cdot (N - 1)}{N - \omega - 2}.$$

In general, then, a stochastic algorithm using this Bayesian estimate should terminate with the number of observed unique local minima ω exceeds this estimate by no more than some tolerance δ . That is, the method should terminate when N and ω satisfy

$$\omega \leq \frac{\omega \cdot (N - 1)}{N - \omega - 2} \leq \omega + \delta,$$

which can be rewritten as a function of ω and δ as

$$N \geq \frac{\omega^2 + \omega}{\delta} + \omega + 2.$$

The tolerance δ provides control over the “rigorousness” of the estimate. Smaller δ provide a greater “guarantee” that the global minimum has actually been found, but this assurance comes at the price of a larger number of trials required for any specific number of unique local minima observed. Conversely, for larger δ (although never greater than 1), fewer trials must be executed for a given number of unique local minima, although too few trials may not provide enough of a “guarantee” that the global minimum has been found. A typical value of $\delta = 0.5$ has been suggested as a practical requirement by Byrd et al [5].

It is important to note that this estimate requires a number of trials N that is *quadratic* in the number of unique local minima ω . Since the number of local minima tends to increase *exponentially* as the number of variables is increased, this estimate can become quite large for relatively simple problems. As a result, Boender and Rinnooy Kan have suggested an alternate criteria which derives from the following theorem:

Theorem 2.2: Let ω be the number of different observed local minima obtained as a result of performing N uniformly distributed random local searches. The optimal Bayesian estimate of the total relative volume of the observed regions of attraction, for $N \geq \omega + 2$, is

$$\frac{(N - \omega - 1) \cdot (N + \omega)}{N \cdot (N - 1)} = 1 - \frac{\omega \cdot (\omega + 1)}{N \cdot (N - 1)}.$$

Each local minimum in an optimization problem has a corresponding “region of attraction,” in the sense that any search begun in this region will result in finding that minimum. Clearly the total space of the feasible region will be covered with such a non-overlapping set of regions of attraction. Hence, this second stopping rule is based on the percentage of the feasible region that it estimates has been observed, rather than being

based directly on the number of distinct observed local minima. Since both of these rules and their associated estimates are strictly probabilistic, there can be no guarantee that the global minimum has been observed in either case, and therefore no reliability-based reason for picking one rule over the other. If t is the relative volume of the feasible region that has been explored so far, this relation can be rewritten as

$$1 - \frac{\omega \cdot (\omega + 1)}{N \cdot (N - 1)} \geq 1 - \left(\frac{\omega + 1}{N - 1} \right)^2 \geq t,$$

As before, this last inequality can be written as a function of ω and t to give an estimate of the number of trials N :

$$N \geq \frac{\omega + 1}{\sqrt{1 - t}} + 1.$$

This estimate, in contrast with the first, is only linear in the number of observed unique local minima. However, because of the constants involved, the first stopping rule will require fewer trials than this one for small numbers of local minima. The point at which it becomes more efficient to use this rule is dependent on the values chosen for t and δ , but for typical values of $t = 0.999$ and $\delta = 0.5$, the graph below (Figure 2.1) shows that if fewer than 16 unique local minima have been observed, the quadratic stopping rule should be used. If 16 or more local minima have been observed, the linear stopping rule should be used.

As the number of unique local minima grows, the advantage in using the second, linear, stopping rule over the first increases dramatically. Using the same tolerances as before, for a problem with 100 unique observed local minima, the quadratic stopping rule

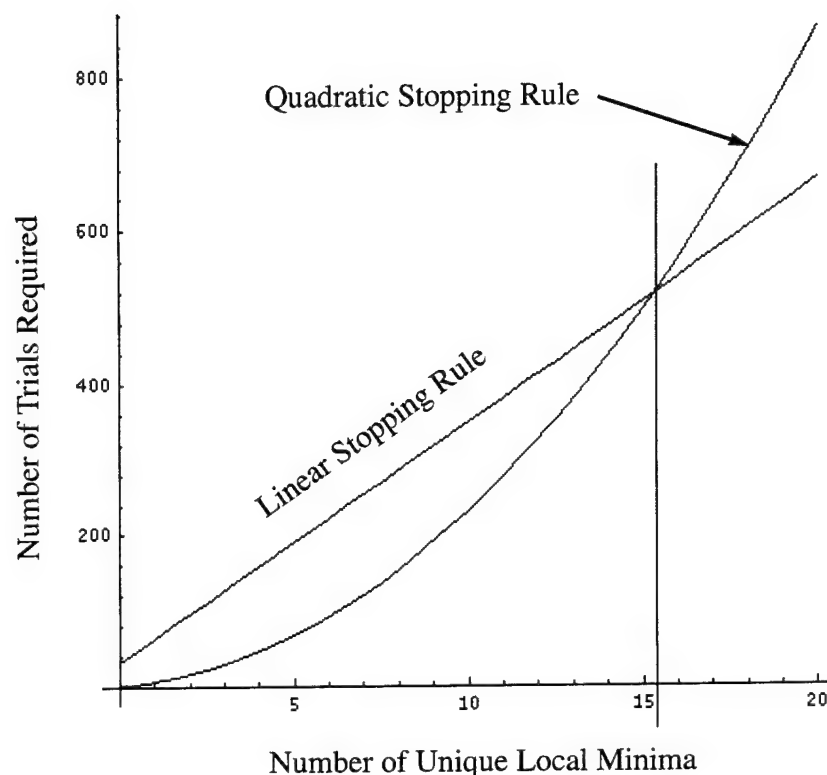


Figure 2.1: Comparison of Stopping Rules

would require 20302 trials, while the linear rule would only require 3195 trials. This is more than a six-fold advantage, and the difference becomes even greater as the size of the problem increases. Figure 2.2 shows the dramatic growth rate of the quadratic stopping rule used in comparison to the linear rule, as ω becomes large.

6. Termination

Once the requirements of either the quadratic or the linear stopping rules are met, the algorithm may terminate, and the global minimum is just the “best” of the local minima discovered. An important point to note is that no matter how tight the tolerances

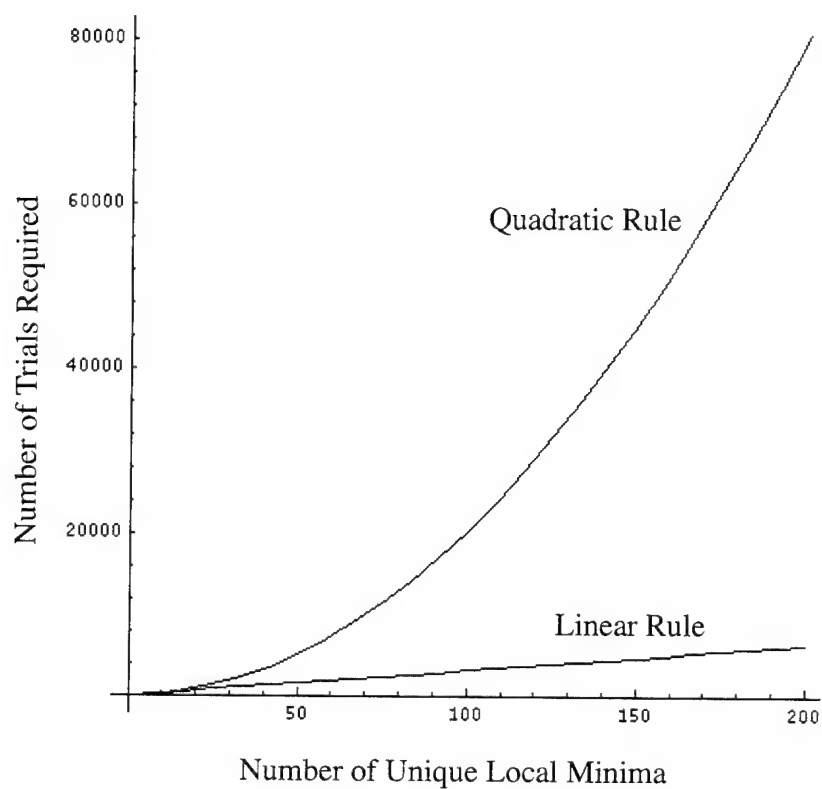


Figure 2.2: Stopping Rule Comparison (large ω)

are set, this algorithm is still probabilistic and is *never guaranteed* to find the true global minimum.

III. Guaranteed Bound Method

In contrast to the stochastic method, the guaranteed bound algorithm provides a *deterministic* way to locate the global optimizer of a linearly constrained indefinite function. It is also guaranteed to converge to the global solution in a finite amount of time, and to any specified tolerance. Although given enough time this method will always find the exact global minimum, it is better suited to applications where the exact answer is not necessary and an answer within some percent of the optimal solution is acceptable. In these cases the guaranteed bound method can be extremely fast.

1. Objective Function Limitations

The first limitation of this algorithm is that the objective function must be separable. That is, the function must be of the form $\varphi(\mathbf{x}) = \sum \varphi_i(x_i)$ where each φ_i is a function of only one variable. For example, $\varphi(\mathbf{x}) = 2x_1^3 + 3x_2 - x_3^2$ is separable, but $\varphi(\mathbf{x}) = x_1x_2 - 3x_1$ is not because the x_1x_2 term involves more than one variable. The other limitation of this method is that the function can not be indefinite in any single dimension. This means that each $\varphi_i(x_i)$ is required to be either convex, concave, or linear. Since many real-world problems can be modelled by separable objective functions without indefinite terms $\varphi_i(x_i)$, these limitations are not a contraindicative restriction on this method.

2. Overview of Algorithm

The guaranteed bound algorithm works by constructing convex underestimators to the objective function over certain subsets of a hyperrectangle containing the feasible region. These underestimators are minimized over the entire feasible region, and the underestimating function values at their minima are compared to the actual function values at the same points. These comparisons give lower and upper bounds on the minimum value of the objective function. This process is repeated on successively smaller subsets of the hyperrectangle using a branch-and-bound paradigm. As the subsets of the hyperrectangle become smaller, the underestimator approximates the objective function better, so the bounds on the global minimum become tighter. This method is guaranteed to eventually converge to the global minimum in a finite number of steps; however, the number of such steps may be exponential. Hence, some heuristics are incorporated into the algorithm in an attempt to decrease the number of steps required for convergence. Although there is no rigid proof that these heuristics will improve the performance of the algorithm, empirical studies have demonstrated their effectiveness [17].

As an illustration of this algorithm, consider the optimization problem defined by

$$\begin{aligned}
 &\text{minimize } \varphi(\mathbf{x}) = -x_1^2 + 3x_2^2 - 18x_2 \\
 &\text{subject to } x_1 + x_2 \leq 5 \\
 &\quad x_1 \geq 0 \\
 &\quad x_2 \geq 0 .
 \end{aligned} \tag{3.1}$$

This function is depicted without constraints in Figure 3.1, over the region $0 \leq x_1, x_2 \leq 5$.

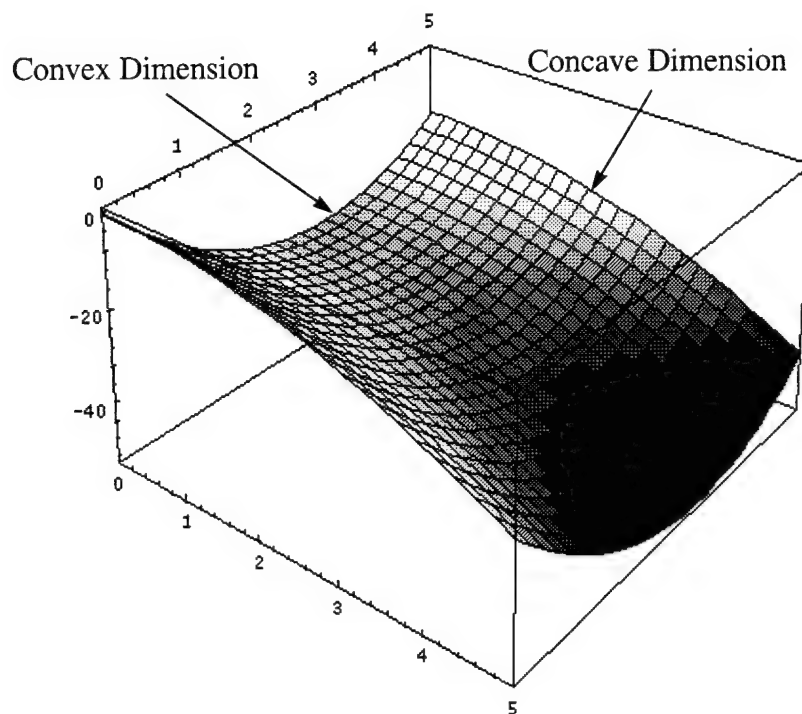


Figure 3.1: Indefinite Function Example

This function is indefinite overall, with one convex dimension (x_2) and one concave dimension (x_1). Figure 3.2 shows the contours of this function, restricted to the feasible region. The darkest areas are the “deepest” parts of the function (those with the lowest function value). As can be seen from this figure, the global minimum subject to the given constraints is at $\mathbf{x} = [3, 2]$. In fact, the point determined by this algorithm is $\mathbf{x} = [2.990, 2.010]$, using a “tolerance” of 0.001 on the value of the global minimum. Since this algorithm is guaranteed to converge in a finite number of iterations, if the tolerance ϵ is set to zero, it will find the exact solution $\mathbf{x} = [3, 2]$. For this algorithm, a solution point \mathbf{x} is defined to be computed within a tolerance ϵ when

$$\left| \frac{\phi(\mathbf{x}) - \gamma}{\phi(\mathbf{x})} \right| \leq \epsilon,$$

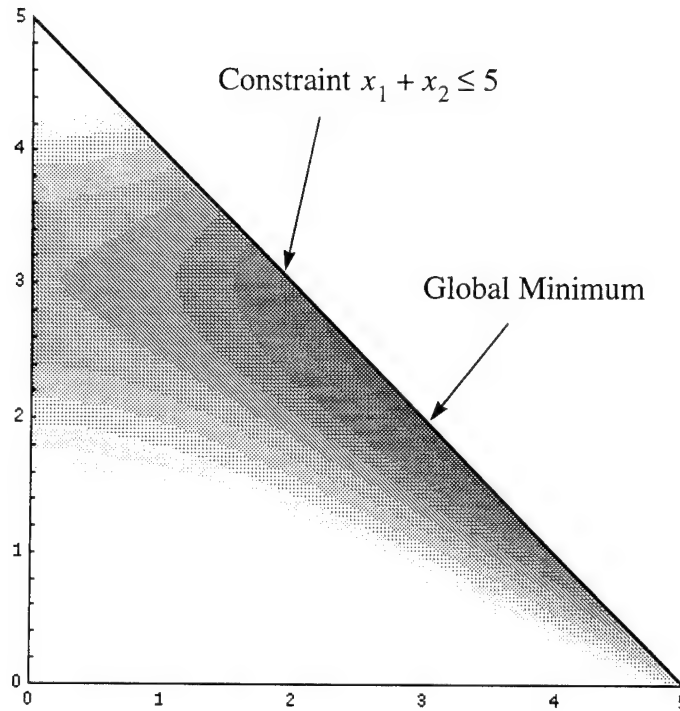


Figure 3.2: Depth of Constrained Function

where γ is any lower bound (ideally, the “best” lower bound) on the objective function ϕ over the feasible region known to contain the global minimum. Furthermore, in order to reduce the error $|\phi(\mathbf{x}) - \gamma|$ as quickly as possible, it is necessary to continuously update the “best” known solution point; that is, the point which gives the minimum value of $\phi(\mathbf{x})$ found so far. This value will hence forth be called $\tilde{\mathbf{x}}$, the incumbent solution.

The first step of this method, as in the case of the stochastic method, is to construct the smallest hyperrectangle which contains the feasible region. This hyperrectangle is again defined as

$$H = \{x_i: l_i \leq x_i \leq u_i, i = 1, 2, \dots, n\},$$

where \mathbf{l} and \mathbf{u} are n -dimensional vectors representing the lower and upper bounds l_i and

u_i for $i = 1, 2, \dots, n$, respectively, on the problem variables \mathbf{x} . Based on this hyperrectangle, a convex underestimator is created which agrees with the indefinite objective function value at each vertex of the hyperrectangle.

Since the indefinite function is separable and each term $\phi_i(x_i)$ of $\phi(\mathbf{x})$ is required to be non-indefinite, creating this convex underestimator is a simple process. Actually, both limitations on this method which were cited earlier are due to this underestimating step, because if the function is not separable, or if one of the individual variables is indefinite, it is very difficult to construct an underestimator for the function. For this algorithm, all concave terms $\phi_i(x_i)$ are underestimated by an affine function $\bar{\phi}_i(x_i)$ which is constructed so that it has the same value as the objective function in that dimension at the endpoints of the hyperrectangle. Hence, $\bar{\phi}_i(x_i) = \phi_i(x_i)$ at the endpoints l_i and u_i of the hyperrectangle along the i^{th} dimension; that is, $\bar{\phi}_i(l_i) = \phi_i(l_i)$ and $\bar{\phi}_i(u_i) = \phi_i(u_i)$. Since $\phi_i(x_i)$ is concave, $\bar{\phi}_i(x_i) \leq \phi_i(x_i)$ over the region of the hyperrectangle and therefore also over the given feasible set. In addition, all convex and linear terms $\phi_i(x_i)$ are underestimated simply by themselves, so $\bar{\phi}_i(x_i) = \phi_i(x_i)$ in those dimensions.

For the example given, the hyperrectangle is computed to be $0 \leq x_1, x_2 \leq 5$. The term $\phi_1(x_1) = -x_1^2$ is concave, so its values at the endpoints of the hyperrectangle must be determined. They are calculated to be $\phi_1(0) = 0$ and $\phi_1(5) = -(5)^2 = -25$. These points are interpolated by the affine function $\bar{\phi}_1(x_1) = -5x_1$. Since the term $\phi_2(x_2) = 3x_2^2 - 18x_2$ is convex, $\bar{\phi}_2(x_2) = \phi_2(x_2)$ and so the total underestimator is therefore

$$\bar{\phi}(\mathbf{x}) = -5x_1 + 3x_2^2 - 18x_2.$$

This underestimator is shown in Figure 3.3.

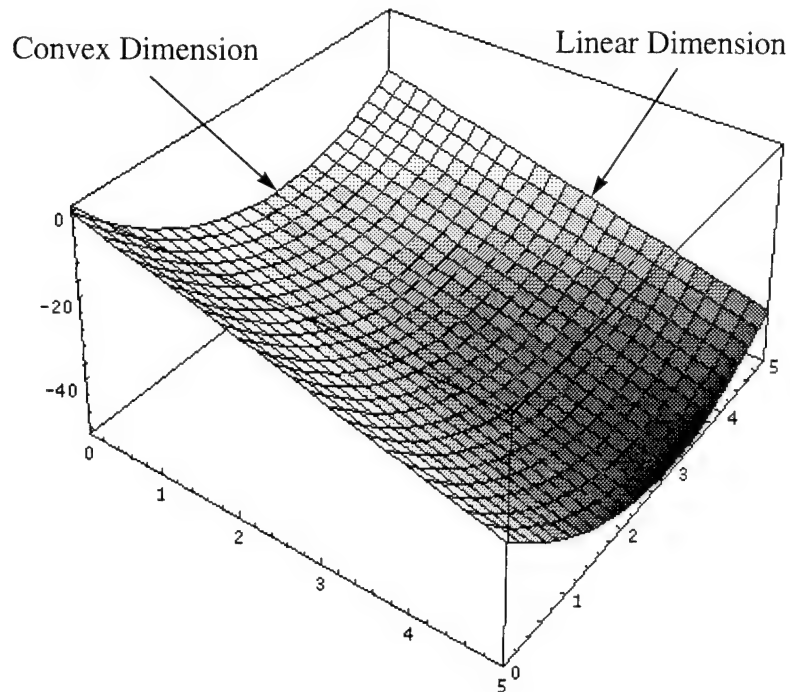


Figure 3.3: Initial Convex Underestimator

This convex underestimator is then minimized over the entire feasible region using the same gradient projection method as the stochastic algorithm. However, since there are no concave variables in the function it is minimizing, this method will always correctly determine the *unique* minimum of the underestimator over the feasible region. For this example the minimum of $\bar{\varphi}$ is discovered at $\bar{\mathbf{x}} = [2.833, 2.167]$. The value of the underestimator at this point is $\bar{\varphi}(\bar{\mathbf{x}}) = -39.083$. Since the function $\bar{\varphi}(\mathbf{x})$ is constructed to underestimate the objective function $\varphi(\mathbf{x})$ throughout the feasible region, and the point $\bar{\mathbf{x}}$ is the minimum point of that underestimator, the value of the objective function inside the feasible region can never be lower than this value; that is, $\bar{\varphi}(\bar{\mathbf{x}}) \leq \varphi(\mathbf{x}^*)$, where \mathbf{x}^* is the yet to be determined global minimum solution. This

provides an initial lower bound γ on the minimum function value. The value of the objective function at the same point is $\varphi(\bar{\mathbf{x}}) = -32.944$, which represents an upper bound on the minimum function value since this point is also inside the feasible region; that is, $\bar{\mathbf{x}} = \tilde{\mathbf{x}}$ is the initial incumbent solution and $\varphi(\mathbf{x}^*) \leq \varphi(\tilde{\mathbf{x}})$. The relative tolerance between these two values is

$$\frac{|\varphi(\tilde{\mathbf{x}}) - \gamma|}{|\varphi(\tilde{\mathbf{x}})|} = \frac{|-39.083 - (-32.944)|}{|-32.944|} = 0.186,$$

which exceeds the tolerance of $\varepsilon = 0.001$ (chosen for this example).

3. Subregion Elimination

At this point the unembellished algorithm would divide the feasible region into two parts and recursively apply itself to each part. By repeating this step, it would iteratively converge on the global minimum. Such an approach represents a pure divide and conquer method which is guaranteed to terminate at the global solution, but only in an exponential number of steps. However, at this point the first heuristic for quicker convergence is applied. For each concave term $\varphi_i(x_i)$ in the function, two subproblems are constructed and an underestimator for each is created. Each new underestimator differs from the original underestimator $\bar{\varphi}(\mathbf{x})$ in only one term, the one involving $\bar{\varphi}_i(x_i)$. In the first subproblem this term is replaced by a different affine function so that it agrees with the objective function $\varphi(\mathbf{x})$ at the lower bound l_i of the hyperrectangle and also the midpoint of the lower and upper bounds, $\frac{l_i + u_i}{2}$. The underestimator for the other subproblem is similar except that it replaces $\bar{\varphi}_i(x_i)$ with an affine function which agrees

with $\phi_i(x_i)$ at the upper bound u_i of the hyperrectangle and again at the midpoint. Effectively, this splits the feasible region into two halves for each dimension in which $\phi_i(x_i)$ is concave. These two halves are underestimated independently, creating $2c$ subproblems, where c is the number of concave terms in the function.

In this example, $c = 1$ since $\phi_1(x_1)$ is the only concave term. For the two subproblems created in this step the midpoint required is $\frac{5-0}{2} = \frac{5}{2}$ and the corresponding function value is $\phi_1(5/2) = -\left(\frac{5}{2}\right)^2 = -\frac{25}{4}$. Using this information, the two underestimators can be constructed. For the first, $\bar{\phi}_1(x_1)$ is replaced by $\bar{\phi}_1^{(1)}(x_1) = -\frac{5}{2}x_1$, so that

$$\bar{\phi}^{(1)}(\mathbf{x}) = -\frac{5}{2}x_1 + 3x_2^2 - 18x_2. \quad (3.2)$$

For the second, $\bar{\phi}_1(x_1)$ is replaced by $\bar{\phi}_1^{(2)}(x_1) = -\frac{15}{2}x_1 + \frac{25}{2}$, so that

$$\bar{\phi}^{(2)}(\mathbf{x}) = -\frac{15}{2}x_1 + 3x_2^2 - 18x_2 + \frac{25}{2}. \quad (3.3)$$

Once these $2c$ underestimators are constructed, each one is minimized over the entire feasible region. If the value of the underestimator $\bar{\phi}^{(i)}$ at its minimum $\bar{\mathbf{x}}^{(i)}$ is greater than the function value at that same point, then the section of the feasible region underestimated by $\bar{\phi}^{(i)}$ can be removed from further consideration. This subregion elimination requirement is

$$\bar{\phi}^{(i)}(\bar{\mathbf{x}}^{(i)}) > \phi(\bar{\mathbf{x}}^{(i)}). \quad (3.4)$$

For example, consider the function $\phi(x) = -x^2 + 1$ over the range $[-1, 5]$, as shown in Figure 3.4. This graph also shows the underestimator for the entire feasible region, $\bar{\phi}(x)$,

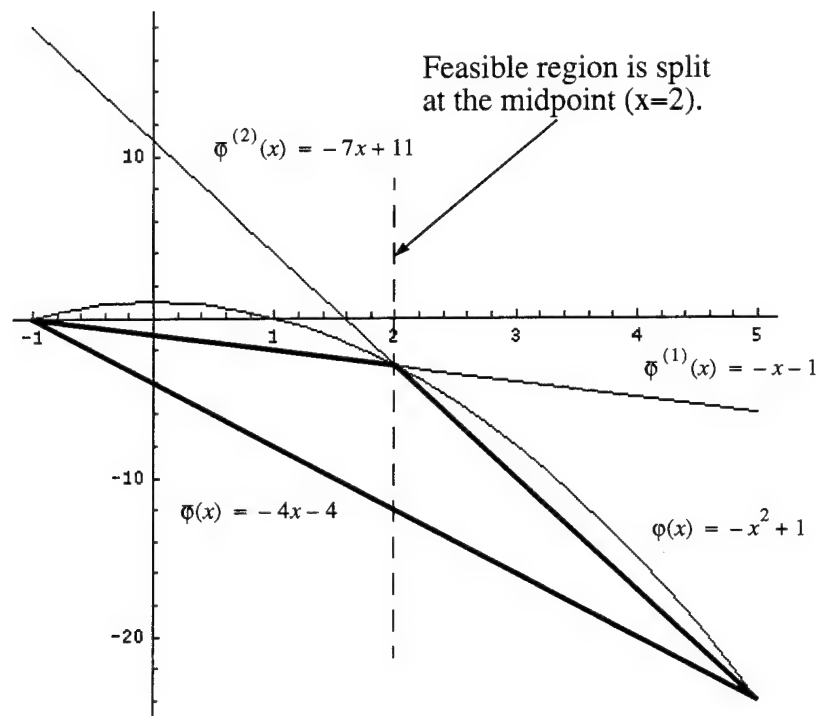


Figure 3.4: Example of Subregion Elimination

and the underestimators for each half, $\bar{\varphi}^{(1)}(x)$ and $\bar{\varphi}^{(2)}(x)$, when the feasible region is split at the midpoint. It is clear from Figure 3.4 that the minimum over the entire feasible region for *each* of these underestimators is at $\bar{x} = 5$. For the first subproblem, the value of the underestimator $\bar{\varphi}^{(1)}(x)$ at the minimum is greater than the function value at that point

$$\bar{\varphi}^{(1)}(\bar{x}) = \bar{\varphi}^{(1)}(5) = -6 > -24 = \varphi(5) = \varphi(\bar{x}).$$

Because of this, the global minimum can not be in the left half of the feasible region, and so that half can be eliminated from further consideration. The algorithm must however continue to search for the global minimum in the right half of the region.

Similarly, when the two underestimators (Equation 3.2 and Equation 3.3) for the function in Figure 3.1 are minimized over the entire feasible region, they give the

solutions $\bar{\mathbf{x}}^{(1)} = [2.417, 2.583]$ and $\bar{\mathbf{x}}^{(2)} = [3.250, 1.750]$. The values of the underestimators at these points are $\bar{\varphi}^{(1)}(\bar{\mathbf{x}}^{(1)}) = -32.521$ and $\bar{\varphi}^{(2)}(\bar{\mathbf{x}}^{(2)}) = -34.188$, and the corresponding values of the function at these points are $\varphi(\bar{\mathbf{x}}^{(1)}) = -32.319$ and $\varphi(\bar{\mathbf{x}}^{(2)}) = -32.875$. Although in this case the values are all close, neither of the two subproblems satisfied the subregion elimination requirement, Equation 3.4. Therefore, the algorithm must continue in the divide and conquer style by adding a new constraint to split one of the dimensions, and then recursively attempt to find the global minimum in both new regions.

4. Updating the Bounds

In order to terminate at an “ ϵ -approximate” solution in a reasonable amount of time, it is essential that the error bound $|\varphi(\tilde{\mathbf{x}}) - \gamma|$ be as tight as possible at every step. Hence, the incumbent solution $\tilde{\mathbf{x}}$ must always be the best known function value found so far. Updating the lower bound γ at each step is more complex since this value must, of necessity, satisfy $\gamma \leq \varphi(\mathbf{x}^*)$, where \mathbf{x}^* is *unknown*! However, since $\bar{\varphi}(\mathbf{x}) \leq \varphi(\mathbf{x})$ for all \mathbf{x} in the feasible region known to contain \mathbf{x}^* (recall that this region is iteratively reduced by the heuristic step, but will always contain \mathbf{x}^*), then $\bar{\varphi}(\bar{\mathbf{x}})$ is an obvious lower bound on $\varphi(\mathbf{x}^*)$. Furthermore, since $\bar{\varphi}_i^{(1)}(x_i)$ and $\bar{\varphi}_i^{(2)}(x_i)$ are constructed to underestimate $\varphi_i(x_i)$ over their respective halves of the hyperrectangle (and hence the feasible space) for all concave terms i , then *both*

$$\bar{\varphi}^{(1)}(\bar{\mathbf{x}}^{(1)}) = \sum_{j \neq i} \bar{\varphi}_j(\bar{x}_j^{(1)}) + \bar{\varphi}_i^{(1)}(\bar{x}_i^{(1)})$$

and also

$$\bar{\varphi}^{(2)}(\bar{\mathbf{x}}^{(2)}) = \sum_{j \neq i} \bar{\varphi}_j(\bar{x}_j^{(2)}) + \bar{\varphi}_i^{(2)}(\bar{x}_i^{(2)})$$

are valid underestimators over their particular halves of the feasible region. Thus, the

$$\min \{ \bar{\varphi}^{(1)}(\bar{\mathbf{x}}^{(1)}), \bar{\varphi}^{(2)}(\bar{\mathbf{x}}^{(2)}) \}$$

is a valid lower bound over the whole feasible region under consideration. Since this holds for *all* concave terms i , then in fact the best lower bound at each iteration can be updated using

$$\gamma = \max_{\text{concave terms}} \{ \min \{ \bar{\varphi}^{(1)}(\bar{\mathbf{x}}^{(1)}), \bar{\varphi}^{(2)}(\bar{\mathbf{x}}^{(2)}) \} \}.$$

Hence, at the conclusion of the $2c$ convex underestimators, the lower bound γ and the upper bound $\varphi(\tilde{\mathbf{x}})$ are updated and the stopping criteria $\left| \frac{\varphi(\tilde{\mathbf{x}}) - \gamma}{\varphi(\tilde{\mathbf{x}})} \right| \leq \varepsilon$ is checked to determine if the algorithm may be terminated.

5. Divide and Conquer

Two methods are used by this algorithm to determine which dimension should be split. The first approach is to compute the value of the underestimator for the entire feasible region at the current incumbent solution in each dimension *individually*, and compare those values to the value of the function in each dimension at that point. Since the function is separable, this is a simple process. Hence, if $\tilde{\mathbf{x}}$ represents the current incumbent solution, then in this example the algorithm computes

$$\bar{\varphi}_1(\tilde{\mathbf{x}}_1) - \varphi_1(\tilde{\mathbf{x}}_1) \text{ and } \bar{\varphi}_2(\tilde{\mathbf{x}}_2) - \varphi_2(\tilde{\mathbf{x}}_2).$$

The largest of these differences indicates the side which should be split since it is in that dimension that the greatest “error” occurs between the lower and upper bounds at the best known solution. This is known as the “max gap” rule. The other method for choosing the side is to simply split on the side for which the current bounds on the feasible region are farthest apart. In fact, both of these methods are used in an attempt to converge on the global minimum quickly. Without the inclusion of the second method (the “max side” rule), it is theoretically possible to get “stuck” splitting the same side in an infinite sequence and never converge on the actual global minimum [22]. In general this is *not* a practical problem; however, in the interest of “guaranteeing” finite convergence, it is used approximately 10% of the time. Specifically, the “max side” rule is used every tenth time the algorithm splits to prevent any infinite non-convergent sequences resulting from the use of the “max gap” rule.

After the dimension along which to divide the feasible region has been chosen, the algorithm must decide where in that dimension to split. The obvious answer is to divide at the midpoint, and this works very well [17]. However, there is evidence that splitting instead at the minimizer for the global underestimator for that step will cause the algorithm to converge quicker than splitting at the midpoint [22]. Splitting at this point tends to increase the number of subproblems which can be quickly eliminated, since the underestimators for these subproblems are more likely to meet the condition $\bar{\varphi}(\bar{\mathbf{x}}) > \varphi(\bar{\mathbf{x}})$.

For the example from Equation 3.1, the dimension with the largest difference between lower and upper bounds is the first dimension, so the feasible region is divided

in that dimension (from the "max gap" rule). The location of the incumbent solution point after the first step was $\tilde{\mathbf{x}} = [2.833, 2.167]$ with corresponding function value $\phi(\tilde{\mathbf{x}}) = -32.944$, so the first new problem has the added constraint $x_1 \leq 2.833$. After several subproblem computations, the algorithm determines that the global minimum cannot be in this region, and so this region is eliminated and the second new problem is begun. This problem has the new constraint $x_1 \geq 2.833$. The underestimator for this new feasible region is minimized, obtaining the point $\bar{\mathbf{x}} = [3.306, 1.694]$ with value $\phi(\bar{\mathbf{x}}) = -32.813$. Note that $\bar{\mathbf{x}}$ is a candidate for the global minimum (as are all feasible solutions), but $\phi(\bar{\mathbf{x}}) > \phi(\tilde{\mathbf{x}})$ and so the current incumbent solution does not change.

Since only the first variable is concave, two underestimators are constructed for each half of the feasible region along this dimension. The minimizer for the second underestimator is $\bar{\mathbf{x}}^{(2)} = [3.486, 1.514]$ and its value at that point is $\bar{\phi}^{(2)}(\bar{\mathbf{x}}^{(2)}) = -31.876$, while the function value $\phi(\mathbf{x})$ at that point is $\phi(\bar{\mathbf{x}}^{(2)}) = -32.527$. Since the value of the underestimator is greater than the value of the function at the minimizer of the underestimator, that half of the feasible region can be eliminated, which changes the upper bound on the first variable to $\frac{5.0 - 2.833}{2} = 3.917$. A new underestimator is then constructed for this new feasible region, and after two more subproblems, this dimension can again be reduced to the upper bound $x_1 \leq 3.375$. With this final feasible region the algorithm is able to construct an underestimator so that the relative difference between the computed lower bound γ (which turned out to be $\gamma = -33.018$) and the value of the function at the incumbent solution (which eventually resulted in $\tilde{\mathbf{x}} = [2.990, 2.010]$) is less than the tolerance of $\epsilon = 0.001$.

Once the algorithm has found a point which satisfies the relative tolerance requirement for the global minimum, it outputs both the upper and lower bounds on the value of the global minimum, and also the incumbent solution point. Since the upper and lower bounds on the value of the global minimum satisfy

$$\gamma \leq \varphi(\mathbf{x}^*) \leq \varphi(\tilde{\mathbf{x}}),$$

then

$$\left| \frac{\varphi(\tilde{\mathbf{x}}) - \varphi(\mathbf{x}^*)}{\varphi(\tilde{\mathbf{x}})} \right| \leq \left| \frac{\varphi(\tilde{\mathbf{x}}) - \gamma}{\varphi(\tilde{\mathbf{x}})} \right| \leq \varepsilon,$$

which implies that $\varphi(\tilde{\mathbf{x}}) \approx \varphi(\mathbf{x}^*)$; that is, $\tilde{\mathbf{x}}$ is guaranteed to be a global minimum (to within the tolerance ε). In this example the global minimum point found by the algorithm is $\tilde{\mathbf{x}} = [2.990, 2.010]$ with corresponding function value $\varphi(\tilde{\mathbf{x}}) = -32.9998$, while the exact global minimum is known to be $\mathbf{x}^* = [3, 2]$ with corresponding function value $\varphi(\mathbf{x}^*) = -33.0$.

IV. PVM Network

The Parallel Virtual Machine (PVM) package is a system of library functions which extend the C programming language set of instructions in order to allow a heterogeneous collection of machines to function as a unified whole. It was originally developed in 1989 at Oak Ridge National Laboratory and continues as an ongoing research project. The version of PVM used to implement these algorithms is version 3.3.6, which is available along with supporting documentation via anonymous ftp from `netlib2.cs.utk.edu` in the directory `pvm3` [7].

Software using the PVM system appears to the user to be running on a multiple-instruction, multiple data (MIMD) parallel computer using distributed memory and message-passing based communications. This pseudo-parallel computer is known as a *virtual machine*, and the individual computers running under this system are known as *hosts*.

1. Description of the PVM Package

The PVM package includes functions in the following areas: process control, information, dynamic configuration, signals, option control, and message passing. The process control functions allow a client program to enter and exit the PVM system, as well as start up or kill PVM slave processes on the virtual machine. The information functions provide process identification numbers and other information about processes running on the virtual machine, as well as information about the hosts in the virtual

machine. Client PVM processes can add and delete hosts in the virtual machine using the dynamic configuration routines. This is useful if computational requirements change or if more machines are available at certain times (i.e., weekends, holidays). The signaling functions provide the capability of sending a UNIX signal from one client process to another, as well as requesting the PVM system to notify the client process if certain events occur. The details of the PVM package's operation can be modified by options that are accessed through the option control routines. Finally, the greatest number of PVM functions relate to message passing. These routines manage message buffers, pack and send data, and receive and unpack data.

In addition to this basic library, the user interface part of the PVM package also has a supplemental library of dynamic process group routines, which allow client processes to interact in discrete groups. The second part of the PVM system is a daemon process which must be running on every host in the virtual machine, and which is responsible for communications between the hosts. PVM is designed so that the networks used, as well as the individual machines, can be heterogeneous. The daemon process handles all of this transparently, including data conversion between hosts if they do not use the same integer or floating point representations.

Algorithms designed to run using PVM are also typically written in two parts. The first part is the *master* process. This part is responsible for "spawning" the *slave* processes on hosts in the virtual machine. The master then sends messages to these slave processes, and the slave processes are responsible for interpreting these messages and acting on them.

2. PVM and the Stochastic Method

The stochastic method is well suited for use with PVM, and more generally, for use with any type of MIMD architecture. This algorithm requires many similar yet independent trials to be executed, and then the results of those trials are compared. This process fits well into the master/slave paradigm of PVM. In this implementation, a master process is responsible for the overall control of the program, generation of starting points, comparison of results, and output of the final data.

As soon as the master process begins, it loads the data for the objective function and the constraints, and initializes a random number seed for use by the random number generator. It then starts the slave processes on other hosts in the virtual machine, and passes them the same information it received, along with the random number seed it generated.

The master then begins to produce points and distribute them to the slaves. By initially sending two points to every slave, the master can ensure that the hosts on which the slaves are executing are never idle. The master then enters a loop, waiting for minima to be returned from the slaves. Every time a minimum is returned from a slave, the master immediately sends another point to that slave so it can keep working. The returned minimum is compared against the list of minima which have already been observed in order to determine if it is a new unique minimum. If the point is new, it is added to the list of minima, and the applicable stopping rule is used to determine the new number of trials necessary to terminate. Regardless of whether or not the minimum is new, the algorithm

checks the number of trials that have been generated against the current Bayesian estimate to determine if the termination criteria have been met. This simple procedure is continued until the stopping criteria are met, at which point the slaves are terminated and the best local minimum is declared the global minimum.

By the very nature of this algorithm, since all of the hosts in the virtual machine are continuously busy, the speedup observed while using PVM on N hosts over using a single machine is almost linear in the number of hosts available for the computation. However, for very small problems, the communication overhead between the master and the slaves can actually reduce the performance of the stochastic method. This overhead becomes negligible for larger problems.

Another factor which degrades performance, especially with smaller problems, is the setup time required to initialize the slave processes and initiate communications between the master and the slaves. Table 4.1 shows the relationship of the setup time (in seconds) to the number of variables, with the number of constraints held constant at 20.

Table 4.1: Setup Time for Constant Number of Constraints

Variables (n)	Setup Time (secs)	Run Time (secs)	Total Time (secs)	Percent of Total
10	1.88	2.56	4.44	43.1
15	1.95	4.74	6.69	29.3
20	2.67	14.54	17.21	19.4
25	4.65	46.55	51.21	14.7
30	6.56	279.74	286.30	5.4
40	13.30	391.50	404.80	4.3

Notice that although the setup time increases with increasing n , the percent of total time

spent in the setup routines is constantly decreasing. Table 4.2 illustrates a similar trend, except that the number of variables is now held constant at 25, and the number of constraints is changing. Once again, notice that the percent of the total time spent in

Table 4.2: Setup Time for Constant Number of Variables

Constraints (m)	Setup Time (secs)	Run Time (secs)	Total Time (secs)	Percent of Total
25	7.21	73.07	80.28	16.4
35	6.66	55.91	62.57	12.1
40	7.18	57.05	64.23	11.6
45	5.21	77.19	82.40	6.4

setting up the problem is constantly decreasing. Each time given in Table 4.1 and Table 4.2 is the average of the solution times from a minimum of five test problems. It is clear from the empirical data presented above that the overhead incurred from setting up the PVM package and from communications between the master and the slaves is negligible as the problem size increases.

3. PVM and the Guaranteed Bound Method

While the guaranteed bound method lends itself to MIMD style parallelism using PVM, the resulting performance increase is not as pronounced as it was for the stochastic method. This is a direct result of the fact that it is not possible in general to keep all of the hosts working all of the time during the course of a computation using the guaranteed bound method. This algorithm may do MIMD style tasks both when the feasible region is divided, and when the $2c$ underestimators in each concave dimension are minimized.

Although the calculation of the dual underestimators takes an appreciable amount of time, minimizing each one is relatively efficient, so the communication overhead in a MIMD model would likely dominate this phase. However, the recursive calls generated by a split in the feasible region are more suitable for a MIMD implementation. Each time the algorithm subdivides the feasible region, it adds two nodes to the execution tree. These nodes are sent out to slaves which are not currently busy, and are then solved in parallel on these slaves. When the nodes are solved sequentially, the algorithm may spend valuable processing time fathoming a branch of the solution tree which does not contain the global solution. When implemented in parallel, the algorithm can search both halves of the tree simultaneously, so it will typically find the global minimum more quickly.

V. MasPar MP-1

The MasPar MP-1 is a massively parallel supercomputer which uses large numbers of relatively weak processors to perform large, computationally-expensive tasks quickly. It excels at data-parallel calculations, which are the hallmark of a single-instruction, multiple-data (SIMD) style machine. The SIMD paradigm typically involves a “distributed” memory, which means that each processor has its own cache of local memory in which to store data. In the model on which these algorithms are implemented, there are 4096 processing elements (PEs), and each PE has 64 kilobytes of local memory for its own use. In a SIMD computer, each of the parallel processors must be executing the same instruction at the same time. The only exception to this is that some subset of the processors can be made to execute *no* instruction during a cycle. Because of this synchronicity requirement, these algorithms can not use the same methodology that was used with the PVM package to accelerate their processing and improve their performance.

1. Stochastic Method

In the stochastic method, the trials can not be done simultaneously on different processors, because they would be executing different instruction sequences. Each processor could hold in its own memory all of the information for the individual trials, such as the current point, gradient, and active set of constraints. However, the current point, gradient, and set of active constraints would typically be different for each

processor, so as the individual processors are executing through the algorithm trying to find local minima, different processors will find different minima at different times. Also, many of the instructions used to calculate the descent direction \mathbf{d} are dependent on the number of constraints in the active set, and because each processor would likely have a different set of active constraints, the synchronization between the processors would be lost, along with the possible performance gains from using such a large number of processors.

These restrictions indicate that a different approach to parallelism of the stochastic method is required when it is implemented on the MasPar MP-1. Although the “trial-level” parallelism is not readily adaptable to a SIMD architecture, other parts of the code can be written to execute in parallel. Specifically, the code which calculates the descent direction \mathbf{d} contains many matrix operations. Matrix and vector operations are ideal instructions to execute using a SIMD architecture. Hence, although each trial must be performed serially, the matrix operations can be done quickly in parallel.

This “matrix-level” type of parallelism has been implemented, but the performance is quite poor in comparison to the PVM implementation. The increase in speed that is gained by performing these matrix operations in parallel is offset by the degradation that is a result of only having only a single processor which can execute the remaining serial portions of the code. Table 5.1 shows the results of several tests performed to compare the execution time using the PVM package against the time using the MasPar MP-1. For each problem listed, n and m represent the number of variables and constraints, respectively. “PVM Time” is the time required for the test problem to

execute under PVM (setup times are not included). Likewise, "Seq. MP-1 Time" is the time the same problem required to execute on the MasPar MP-1 (without using any parallelism) and can be compared to the "Parallel MP-1 Time" (the execution time using "matrix-level" parallelism) to show how much the parallelism on the MasPar MP-1 helped to speed up the solution procedure. The "Parallel Percent" column displays the percent speedup that was achieved using the matrix-level parallel code over the strictly sequential implementation on the MasPar MP-1. Both "Ratio" columns show the ratio of the time for the MasPar MP-1 code versions to the time for the same problem when executed under PVM. Even when using the parallel code, the MasPar MP-1 version is an average of 8.2 times slower than the PVM version.

Table 5.1: Comparison of MasPar MP-1 and PVM Speed

Test	n	m	PVM Time (secs)	Seq. MP-1 Time	Ratio	Parallel MP-1 Time	Ratio	Parallel Percent
1	30	50	282.18	5498.11	19.5	2650.84	9.4	51.8
2	30	50	162.49	2603.89	16.0	1258.62	7.8	51.7
3	30	40	101.82	1597.18	15.7	766.03	7.5	52.0
4	30	40	124.07	1570.05	12.7	704.98	5.7	55.1
5	25	15	27.34	482.99	17.7	236.80	8.7	51.0
6	25	15	75.74	1512.11	20.0	760.18	10.0	49.7

A code profiler reveals that about 72.3 percent of the operations executed during a typical run of the stochastic method use matrix computations, which can efficiently be performed in parallel on the MasPar MP-1. Hence, even if these operations could be performed *instantaneously*, there would be a theoretical improvement of only 72 percent

in the speed of execution when using the parallel implementation on the MasPar MP-1. However, Table 5.1 shows that the parallel implementation only managed an average of a 51.9 percent improvement over the non-parallel version on the MasPar MP-1. This is in part due to the fact that those matrix operations cannot be performed instantaneously, and it is in part due to the time spent communicating between the PEs before and after each set of parallel calculations. Even if the full theoretical improvement of 72 percent were achievable, the problems tested would still have executed an average of 4.8 times slower than the equivalent PVM implementation. These initial tests show that the “level of parallelism” inherent in this stochastic method is not well suited to a SIMD style architecture.

2. Guaranteed Bound Method

The guaranteed bound method suffers from the same problems as the stochastic method when using a data-parallel system. The level of parallelism that was used for the PVM implementation is once again not available with the MasPar MP-1, for the same reasons as with the stochastic version. Again, the primary use of data-parallelism with this method is in the gradient projection method when it performs matrix operations to calculate the descent direction \mathbf{d} .

Once again, profiler output of the guaranteed bound method reveals that about 76.5 percent of the time for this algorithm is spent in code performing matrix operations. Since the average decrease in speed when using sequential code on the MasPar MP-1 as compared to PVM is a factor of 16.9, even the theoretical maximum speedup of 76.5

percent when using parallel code instead of sequential MasPar MP-1 code would still leave the MasPar MP-1 implementation approximately 4.0 times as slow as the PVM implementation (on average). Clearly, this method is not suited for use on the MasPar any more than the stochastic method. In conclusion, a data-parallel system does not tend to work well with either of these methods due to the poor match of the “granularity” of parallelism.

3. Maximum Possible Speedup

The following theorem, known as *Amdahl's Law*, shows that the performance of a SIMD style machine is directly related to the fraction of the code which must be done sequentially [1].

Theorem 5.1: Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup S achievable by a parallel computer with p processors performing the computation is bounded above by

$$S \leq \frac{1}{f + \left(\frac{1-f}{p}\right)}.$$

Since both the stochastic and the guaranteed bound algorithms require a minimum of about one-fourth of the code to be executed sequentially, the maximum possible speedup for either method using all 4096 PEs on the MasPar MP-1 is therefore

$$S \leq \frac{1}{0.25 + \left(\frac{0.75}{p}\right)} = \frac{p}{0.25p + 0.75} = \frac{4096}{1024 + 0.75} \approx 4.00.$$

In fact, as is evident from this formula, the maximum speedup using these algorithms will never exceed $\frac{1}{f}$, regardless of how many processors p are used [18]. Figure 5.1 shows

how quickly the maximum speedup diverges from a linear speedup with only ten percent of the code requiring sequential computation. Hence, it is clear from this graph that

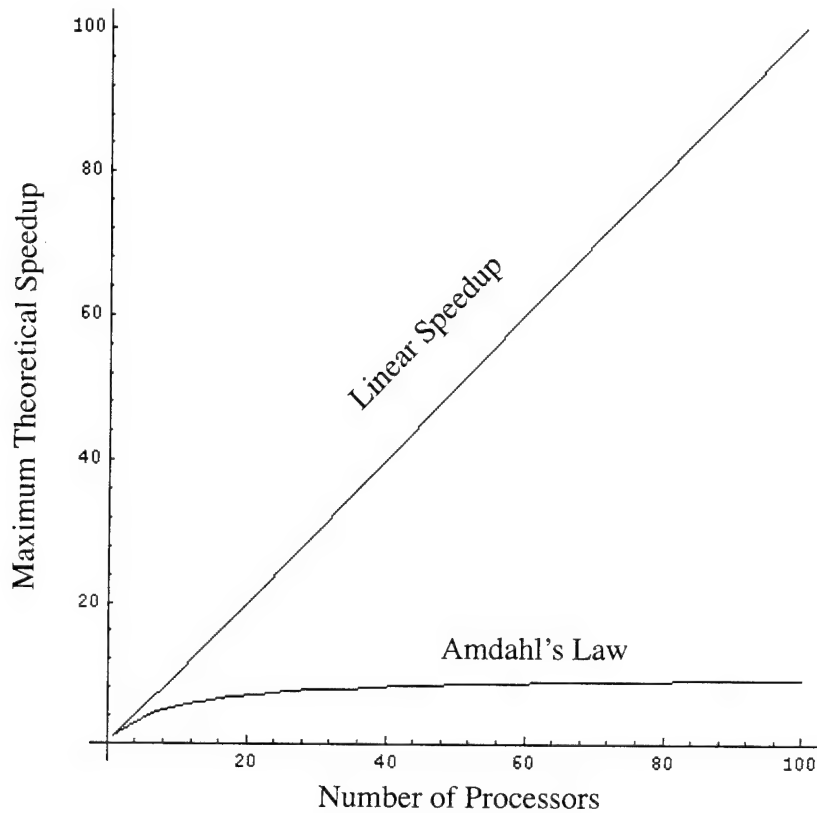


Figure 5.1: Maximum Speedup for 10% Sequential Code

SIMD architectures will never do well with either of these algorithms. The fraction of the code which must be performed sequentially is simply too large.

VI. Stochastic Method Results

The first step in testing these methods must be a check of correctness. Since the stochastic method is only probabilistic, tests were conducted both on problems with known solutions from the literature ([6], [16]), and on larger randomly generated problems with no known solutions. When applied to each of these test problems from the literature, the stochastic method arrived at the same global minimum (or best known solution point) as previously reported. Although these results are evidence that the stochastic method is capable of finding the true global minimum, it must always be remembered that there is *no* guarantee that this will happen in all cases.

1. Stochastic Method Implementation

As mentioned earlier, the version of PVM used to implement this algorithm is version 3.3.6. It is implemented on a network of Sun SPARCstations used in the Computer Science Department of the United States Naval Academy. Since the various machines on the network are not all available for testing purposes all of the time, the testing environment is not constant. On average it consists of ten workstations, mostly SPARC-2s and SPARC-5s along with a single SPARC-20. The master process is always executed on one of the SPARC-2s to ensure that there is no false execution time speedup from using a host with different capabilities for the master.

2. "Random" Test Problems

To fully test this algorithm, a random test problem generator was developed. The random problems generated are quadratic in nature, and of the form

$$\varphi(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{c}^T \mathbf{x}.$$

The matrix \mathbf{Q} and the vector \mathbf{c} are generated through the use of a pseudo-random number generator, which is initialized with some seed at the start of the algorithm. Although this algorithm is capable of solving substantially more complex objective functions, these quadratic functions were found to be robust and difficult enough to fully test this method. To ensure that the objective functions generated are indefinite, the matrix \mathbf{Q} is constructed with both negative and positive eigenvalues. Empirical data from these tests shows that the number of dimensions is evenly split between the concave and convex terms, and that the magnitudes of the negative eigenvalues are comparable to the magnitudes of the positive eigenvalues. Such test problems are known to be among the most difficult in the class of quadratic functions [12].

3. Global Minima

Over 240 random test problems were generated and solved, with problem dimensions ranging from $n = 5$ to $n = 50$ and the number of constraints varying from $m = 5$ to $m = 50$. Although there are many measures of "success" that may be studied, one important characteristic of the stochastic method is its ability to locate the global solution relatively quickly, even though it still has to perform sufficient trials to satisfy

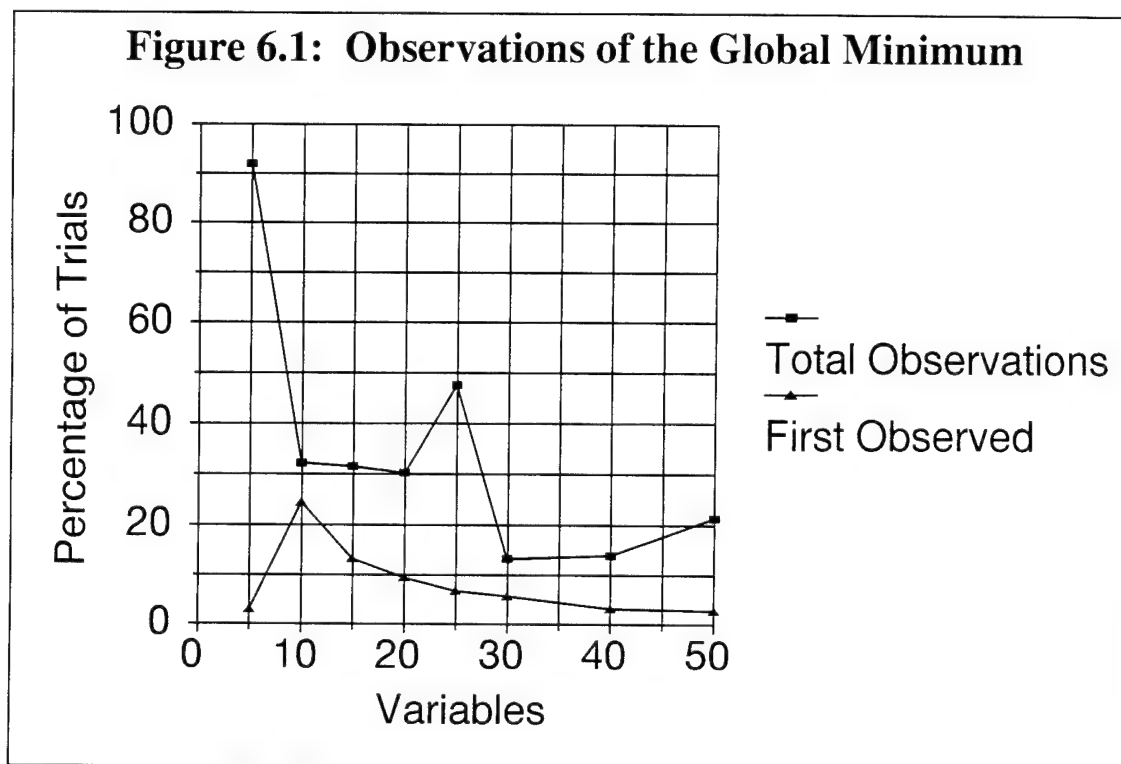
the stopping rules. Table 6.1 shows the relationship between the number of problem

Table 6.1: Global Solution and the Stochastic Method

Variables (n)	Number of Trials	Trial x^* First Seen	Percent of Total Trials	Num. Trials x^* Seen	Percent of Total Trials
5	50.0	1.5	3.0	46.0	92.0
10	207.3	50.9	24.6	66.8	32.2
15	278.1	36.9	13.3	87.9	31.6
20	417.5	39.7	9.5	126.6	30.3
25	429.3	29.2	6.8	204.8	47.7
30	1447.2	82.9	5.7	192.3	13.3
40	1637.2	52.9	3.2	229.2	14.0
50	2543.2	73.2	2.9	545.8	21.5

variables n , and when and how often the global solution was observed (for an average of several test problems). The important features to consider are the trial number on which the solution was first seen as a percentage of the total number of trials (labeled “Trial x^* First Seen” and “Percent of Total Trials”), and the number of observations of the global solution as a percentage of the total number of trials (labeled “Num. Trials x^* Seen” and “Percent Total Trials”). Figure 6.1 displays this graphically. As is clear from the graph, with the exception of the first data point, the trial number on which the global solution is first found, as a percentage of the total number of trials, is a decreasing function. The percentage of trials on which the solution is observed also seems to be decreasing, but the data is not conclusive enough to make a general statement.

It is also important to note that in the 240 test problems which were executed for this data, 94 found the global minimum on the first trial, and 174 (72.5%) found it within



the first five percent of the trials. This demonstrates the fact that if there were sufficient conditions which could be efficiently checked to determine whether a given local minimum is also global, then the stochastic algorithm could terminate much sooner. Unfortunately, it is known that while such conditions exist, they can not be checked *efficiently*. In fact, it turns out that the problem of simply checking such a condition is as hard as solving the original problem [15]! In addition, it is not appropriate to simply ease the tolerances on the stopping rules, because for some of the tests, the global minimum was found very late in the process. Specifically, about 4.5% of the test problems resulted in the global solution being found during the last half of their trials, and in four cases it was found within the last ten percent of the trials.

4. Largest Problems Solved

There are several categories of "largest problem solved" that can be considered. The most useful (although not necessarily the "hardest") is the problem with the greatest number of variables and constraints. For the stochastic method, the largest such problem solved had 50 variables and 50 constraints and required 2034 seconds. Another category is the problem which found the most minima and executed the most trials. By this measure, the largest such problem had 30 variables and 40 constraints, found 547 local minima, and executed 17,330 trials in 4813 seconds. The final category of "largest" problem is the one which required the most time. This problem had 40 variables and 35 constraints, and required 8402.2 seconds (2 hours, 20 minutes) to find the global solution. These three "largest" problems are summarized in Table 6.2.

Table 6.2: Statistics for Three Largest Problems

Variables (n)	Constraints (m)	Setup Time (secs)	Run Time (secs)	Local Minima	Trials
50	50	198.4	2033.6	13	500
30	40	12.6	4812.8	547	17330
40	35	24.8	8402.2	282	8950

5. Local Minima

The number of unique local minima observed is a very important factor which limits the size of problem that this method can handle. Except for the initial setup time, which becomes negligible for larger problems (see Figure 6.4), the number of distinct local minima observed is linearly related to the number of trials that must be performed,

except for small numbers of minima. Although the theoretical relationship between the number of distinct local minima and the number of variables in the problem is known to be exponential, this behavior has not been observed in the test problems, as shown in Figure 6.2. As the number of problem variables increases, the number of observed local

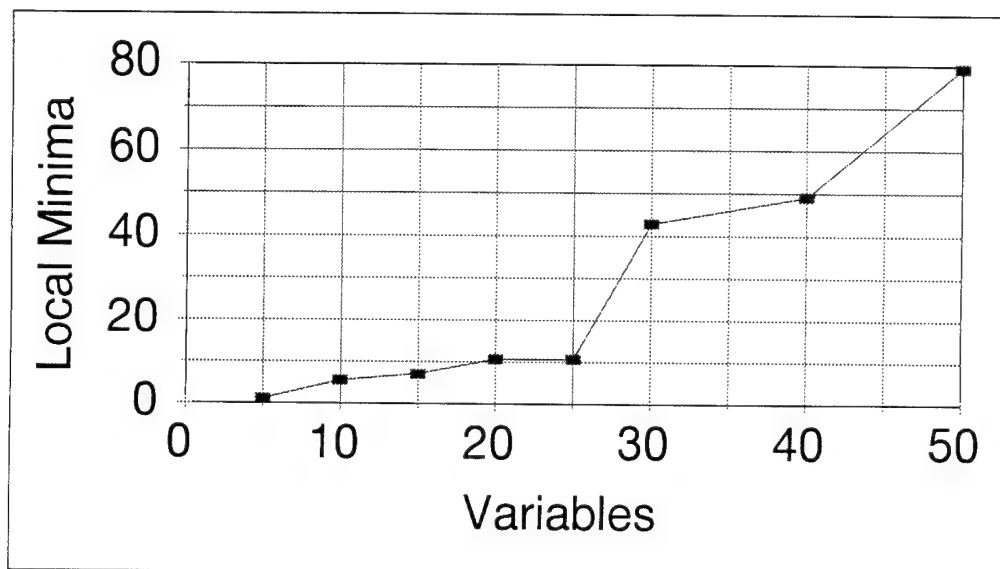


Figure 6.2: Number of Observed Local Minima

minima, and therefore the “hardness” of the problem, increases, but at what appears to be a polynomial (possibly quadratic) rate.

6. Eigenvalues and Problem Difficulty

There are several factors which affect the difficulty of any specific test problem. As already discussed, one of these is the number of local minima observed. Another is the number of variables and the number of constraints defining the problem. In addition to increasing the number of local minima observed, both larger numbers of variables and larger numbers of constraints adversely affect the speed at which the matrix operations

may be performed, which in turn are necessary for calculating the descent direction \mathbf{d} . A third factor which affects the difficulty of the problem is the “curvature” of the concave terms of the function, in this case the magnitude of the negative eigenvalues for the problem. As these values are increased, it becomes more difficult to locate the global solution to the problem ([13], [14], [8], and [23]). Figure 6.3 shows the relationship

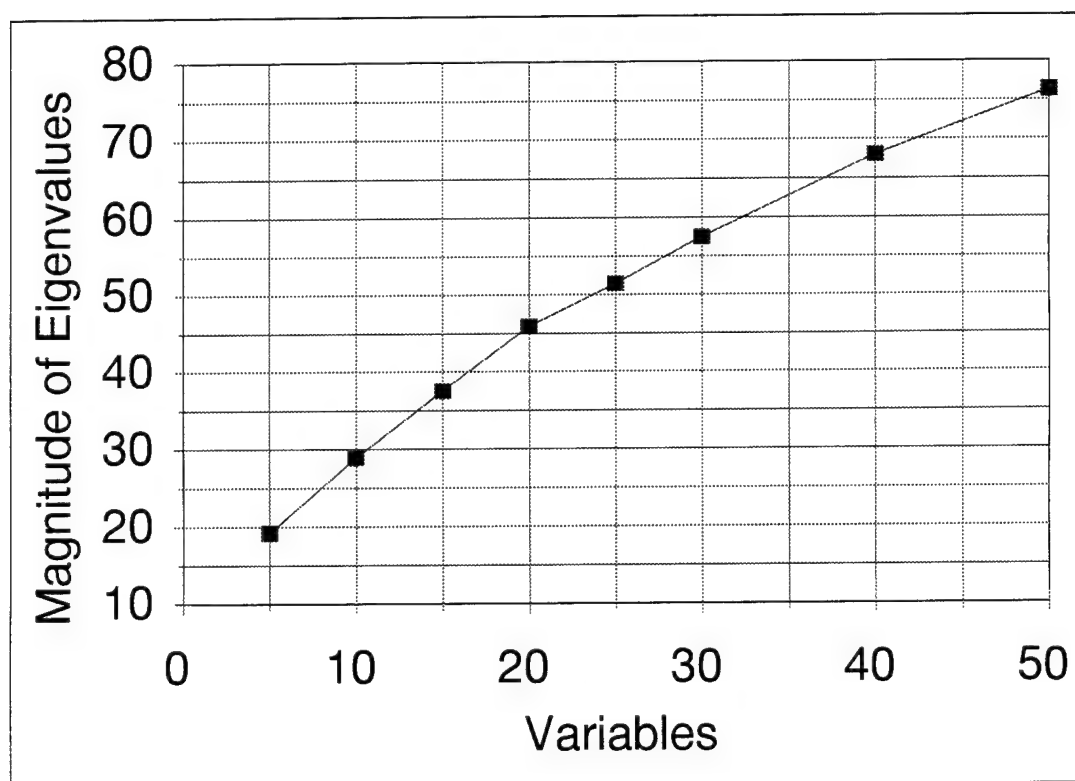


Figure 6.3: Magnitude of the Concave Eigenvalues

between the number of variables in the test problems and the magnitude of the concave eigenvalues for those problems. The increase in the magnitude of the eigenvalues as the number of variables increase shows why larger problems are so much more difficult to solve. All three previously discussed factors, which contribute to the difficulty of the

problem, become more significant as the number of problem variables increases; hence, it is expected that the time required to solve larger problems will grow rapidly.

7. Time of Execution

In fact, the time required to find the global minimum *does* tend to increase dramatically as the number of variables increases. Based on an analysis of the empirical data, and on a study of the actual computations required, the time required for each individual trial is estimated to be bounded above by a polynomial of degree no greater than four (see Figure 6.5). Furthermore, the number of unique local minima, and hence the number of trials required, is *theoretically* exponential in the number of problem variables, so it would be expected that the total time for this method would also be exponential in the number of problem variables. However, this behavior has not been observed. Figure 6.4 shows the relationship, which appears to be polynomial (of degree no more than $O(n^5)$), between execution time and the number of variables for the test problems.

8. Number of Trials

As was previously mentioned, the number of trials is nearly linear with respect to the number of unique observed local minima. This is due to the fact that after about 16 minima have been observed, the algorithm begins to use the linear stopping rule. Hence, the only cases for which the number of trials is not linearly related to the number of local minima is those cases where fewer than 16 local minima were observed during the

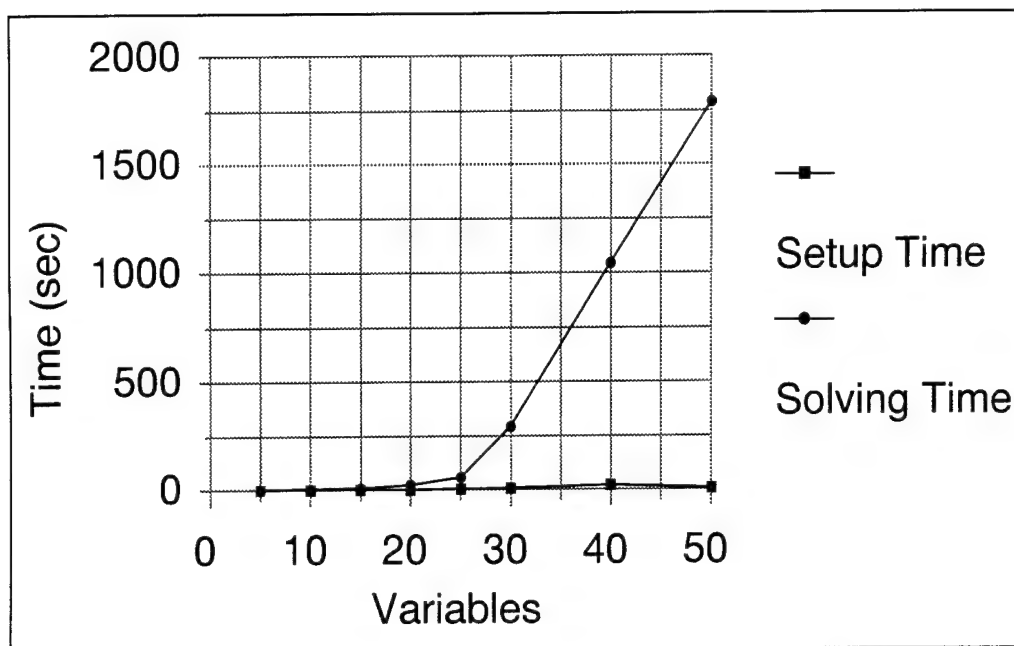


Figure 6.4: Setup Time and Run Time

program run. Table 6.3 catalogs the number of minima and trials for different numbers of variables, as well as the ratio of these two values.

Table 6.3: Number of Trials Required

Variables (n)	Number of Minima	Trials Required	Ratio
5	1.2	50.0	41.7
10	5.6	207.3	37.0
15	7.2	278.1	38.6
20	10.7	417.5	39.0
25	10.8	429.3	39.7
30	42.9	1447.2	33.7
40	49.1	1637.2	33.3
50	79.4	2543.2	32.0

9. Time per Trial

Since the time of execution for any individual test problem is heavily dependent on the number of local minima for that problem, and since even for problems of similar sizes this number varies substantially, a better “measure” is needed to compare the speed of this algorithm on different test problems. One such measure is the time per trial, which is calculated by dividing the run time of the problem (setup time is not included) by the number of trials executed. This value increases with increasing numbers of variables, for a constant number of constraints, and also increases with increasing numbers of constraints, for a constant number of variables. The graph in Figure 6.5 shows this relationship.

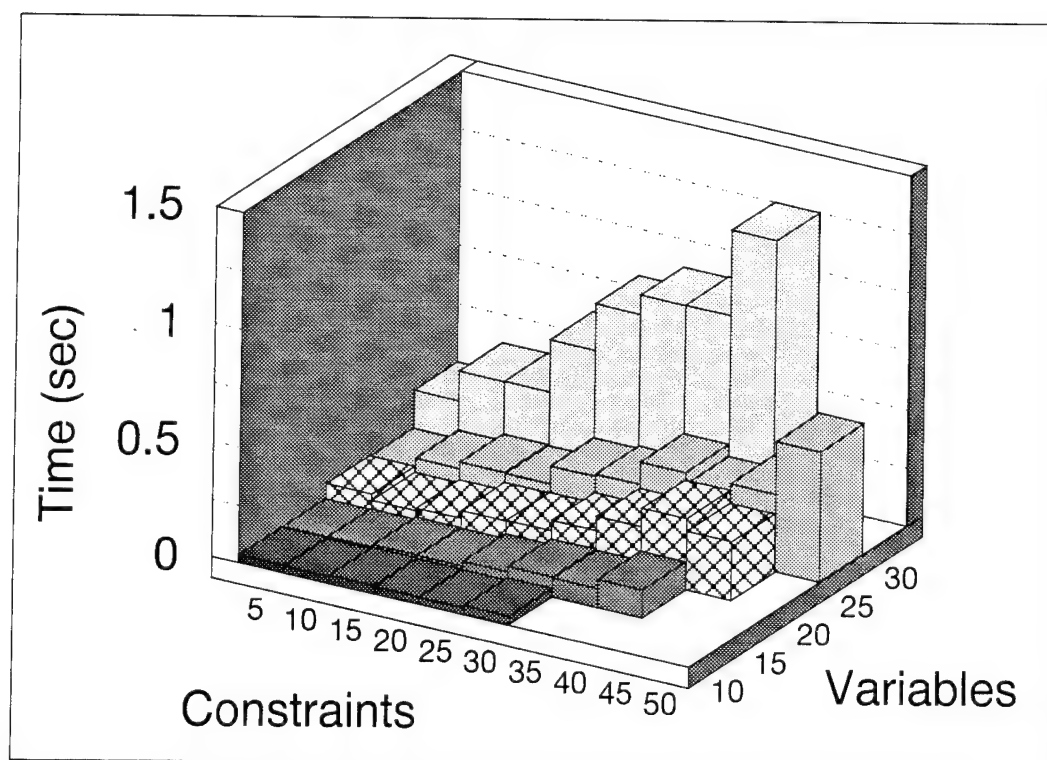


Figure 6.5: Time per Trial for Different Size Problems

10. Concluding Remarks

From these test problems it is evident that although an exponential increase in time is expected as the number of variables increases, the observed behavior is actually polynomial. This result gives hope that larger problems may not be entirely out of reach for methods like this stochastic algorithm. It has also been shown that the time required to set up the problem becomes negligible for larger problems. Since the stochastic method has no guarantee that the global minimum has been found, this method is well suited for problems where the *value* of the global minimum is known a priori, but the *global solution point* is not. In this case, the known value can be checked against the value of the best minimum found by this method to determine if it is in fact the global minimum. This method is also uniquely suited to those cases when all the local minima (or a large majority of them) are desired. And finally, it has been shown that this method does not have the proper level of parallelism to match well with a SIMD style machine; however, it is ideally suited for implementation on a MIMD machine.

VII. Guaranteed Bound Method Results

The guaranteed bound method was also implemented using the Parallel Virtual Machine package, version 3.3.6, on the network of Sun workstations in the Computer Science Department of the U.S. Naval Academy. The same test problem generator was used to construct the problems, so the discussion in Section 6 of Chapter VI on the difficulty of the problems due to the concave eigenvalues is applicable to this method as well.

1. Sequential Execution Time

Clearly, the running time of this method is dependent on the size of the problem tested. This problem size is based not only on the number of variables n , but also on the number of constraints m ; the theoretical and empirical basis for representing the problem size this way is discussed in Section 2. Figure 7.1 shows the relationship between time and problem size, where each data point is an average running time for a number of test runs (between five and ten). Not enough data is available to conclusively determine whether the curve that best fits these points is polynomial or exponential; however, the best fit exponential curve to this data is only $1.18^{(n+m)}$, which is a much slower growing exponential than the theoretical worst case, which would have a base of 2. The best fit single-term polynomial for these data points is approximately $0.0004 (n+m)^4$, which is also a very slowly growing quartic.

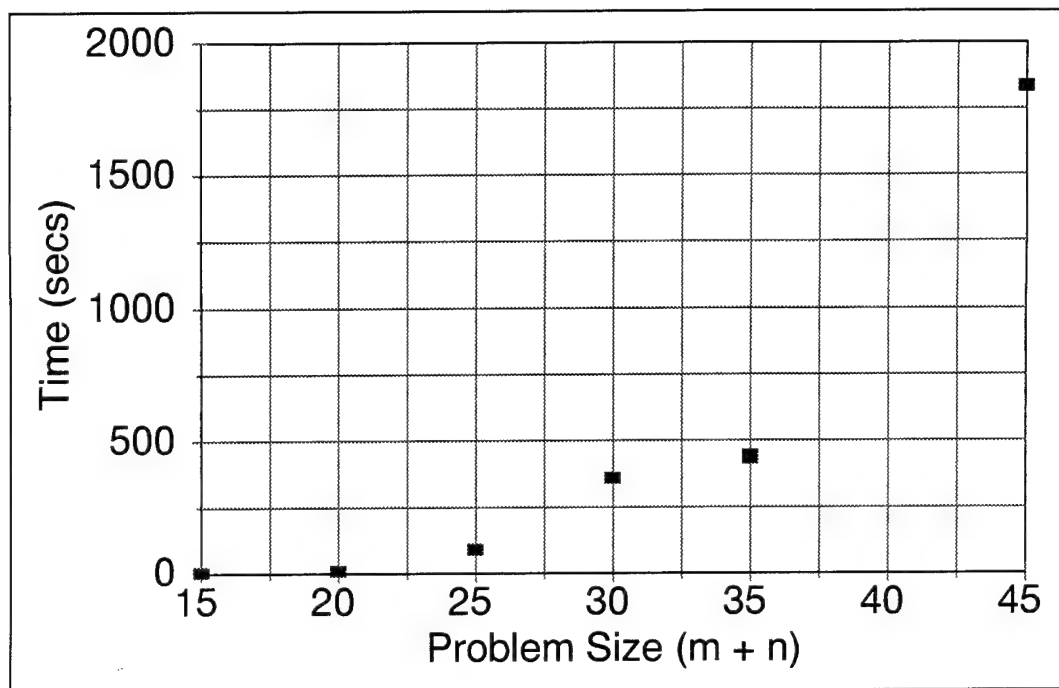


Figure 7.1: Sequential Run Time vs. Problem Size

2. Constraint Dependence

The next result considered is the dependence of the execution time on both the number of variables and the number of constraints. Figure 7.2 shows two sets of data points for a constant number of constraints m . The first set is depicted by the triangles, and is approximated by the quadratic function $335n^2$. The second set of data points is depicted by the squares, and is approximated by the function $83.75n^2$. All of the results in this graph are for execution times using the PVM network, and each data point is again the average of several test runs (again from five to ten). This graph clearly shows a strong dependence of the execution time on both the number of variables and the number of

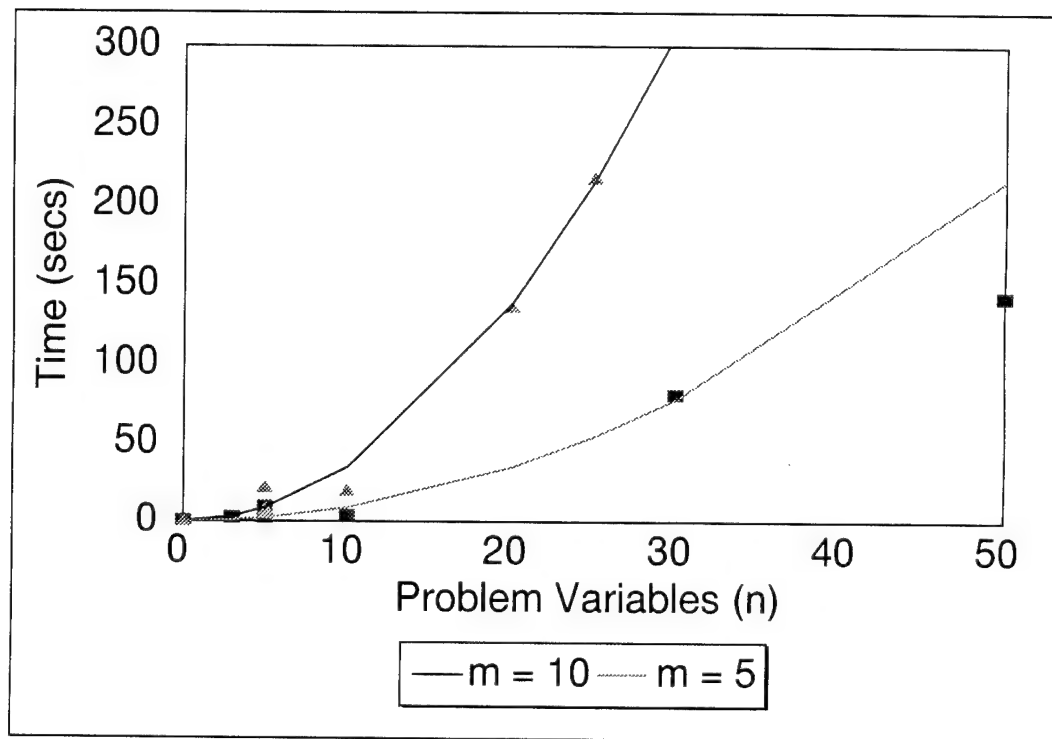


Figure 7.2: Effect of Constraints on Running Time

constraints. It also shows the nature of that dependence, since both function approximations shown in the graph can be thought of as a *single* function of both m and n , specifically $3.35n^2m^2$. The theoretical justification for this function is as follows. The majority of work done in minimizing the underestimators (75 to 85 percent) involves the matrix operations performed to calculate the projected gradient for the new search direction. These operations require time $O(n^2m)$. The remaining factor of m comes from the difficulty of the problem. As the number of constraints increases, it becomes more difficult for this method to tighten the bounds on the global minimum, which require more splits of the feasible region.

3. Parallel Execution Time

The main result of interest in these trials is the execution time for a wide range of problem sizes and an estimation of the complexity of this algorithm. Figure 7.3 shows the

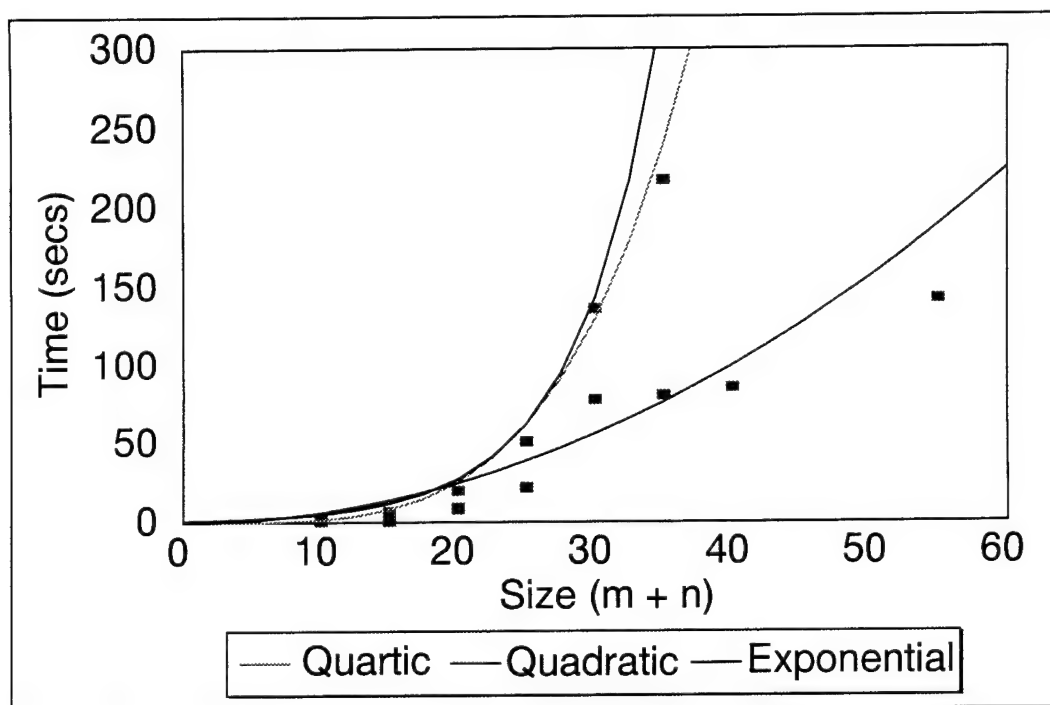


Figure 7.3: Execution Time vs. Problem Size

data points for all tests performed for the guaranteed bound method using the PVM network. As in both previous graphs, each data point is an average of between five and ten actual tests for which the number of variables and the number of constraints was held constant. Three functions have been plotted on the graph; the upper two (quartic and exponential) are approximations of an upper bound on the complexity of this algorithm, and the lower function (quadratic) is an approximation of the running time in the average case. The theoretical justification for the quartic approximation to the upper bound was

presented in the previous section, and the justification for an exponential upper bound is clear, since in the worst case *any* branch-and-bound method must be exponential. However, as in the case of the sequential running times, this exponential is also much slower growing than the theoretical worst case. Also, the quadratic approximation for the average case gives hope that some larger problems may be solved in a reasonable amount of time.

4. Concluding Remarks

As can be seen from Figures 7.1 and 7.3, the speedup gained by using the PVM network varies between three and five. Since the number of machines used in the network averaged ten, the efficiency (i.e., speedup / processors) is only around 50%. However, this is still a significant improvement over using a sequential algorithm on a single machine, since the total "wall-clock" time is greatly reduced.

Although both the sequential and parallel running times can be bounded above by either a polynomial or an exponential approximation, even the exponential approximation does not approach the theoretical worst case. Hence, it is hoped that problems of moderate size representing actual practical applications can be solved reasonably quickly.

Finally, it is important to note the strong dependence of the difficulty of the problem (and hence the running time) on the number of constraints. For many methods, the number of variables is the dominating factor, but in this case the effect on difficulty appears to be evenly spread between the variables and the constraints.

VIII. Conclusions

1. General Notes

It has been noted that neither of these approaches to global optimization of linearly constrained functions can be implemented well on a SIMD style machine. However, both work well under a MIMD paradigm, such as the implementations using the PVM package. The stochastic method acquires a nearly linear speedup in its time of execution based on the number of hosts on which it is simultaneously executing.

Although theoretical work indicates that the problem of finding the global minimum of indefinite objective functions becomes exponentially more difficult as the number of variables increases, the observed behavior based on computational results does not bear this out. In fact, the time of execution for the stochastic method based on the number of variables is *polynomial* of no more than degree five.

2. Future Work

There are several areas in which the stochastic method can be improved. One is in its treatment of bounds as constraints. The current gradient projection method calculates the new descent direction using matrix manipulations on the set of active constraints. For that direction to be restricted from violating any bounds, they must be formulated as additional constraints. This forces the algorithm to do extra work calculating the new direction. If the gradient projection method can be modified to first perform the matrix

calculations, and then include the bounds in a subsequent step, the descent direction could be computed more quickly. This would also benefit the guaranteed bound method, since it also used the same nonlinear programming code to minimize its convex underestimators.

Another way to improve the stochastic method would be to develop early stopping criteria which would allow the algorithm to terminate before satisfying either of the stopping rules, based on some other condition. However, this would not be useful for problems in which a goal is to determine a large majority of the local minima.

The guaranteed bound method can be improved by modifying the domain splitting step. The current method of splitting at the best incumbent point results in the region being split into two parts (bipartite splitting), hence the algorithm can at most throw away one half of the feasible region at each step. Splitting into more than two pieces should be studied, in an effort to remove more of the feasible region at each iteration.

Finally, since it is clear that neither of these methods is well suited for implementation on a SIMD machine, other algorithms should be designed with a SIMD implementation in mind. Instead of reverse-engineering a sequential algorithm into a parallel version, other algorithms should be written from the start to utilize the data-parallel paradigm of a SIMD style machine. This is the only way that the full power of a massively parallel machine such as the MasPar MP-1 will be harnessed.

Appendix A: Sample Program Output

1. Stochastic Method

This is an example of the output produced by the stochastic method, running on a virtual machine under PVM, with ten hosts:

Random seed: -798256808

```
*****
***** Run-time Status *****
*****
```

Trial	Minima	Trials needed
-----	-----	-----
1	1	8
2	2	18
8	3	32
18	4	50
19	5	72
98	6	98
171	7	128
331	8	162
345	9	200
393	10	242

Method: Stochastic (Gradient Projection)

Random seed: -798256808
 Setup time: 112.789 seconds
 Run time: 1214.644 seconds

Variables: 50
 Constraints: 90
 Eigenvalues:
 negative: 26 low: -78.72
 positive: 24 high: 77.20

Minima found: 10
 Total trials: 500

Best Minima: -3664979.341
 Trial: 8
 Count: 130

Location: (0, 0, 0, 0, 0, 0, 119.85, 0, 0, 90.99, 49.70, 241.23, 0, 0, 128.76, 63.92, 97.29, 0, 0, 121.34, 0, 0, 53.19, 0, 161.40, 0, 114.67, 71.21, 68.40, 58.23, 306.85, 0, 75.62, 0, 0, 0, 68.56, 140.01, 0, 0, 0, 10.33, 0, 0, 0, 25.19, 0, 0, 97.49, 69.57)

Process 1835016 at dawson 28 points
 Process 1572876 at haney 43 points
 Process 786439 at topaz 19 points
 Process 786440 at topaz 21 points
 Process 1572874 at haney 35 points
 Process 1572875 at haney 50 points
 Process 2097162 at drucker 21 points
 Process 2359303 at sigsauer 36 points
 ...

2. Guaranteed Bound Method

This is an example of the output produced by the guaranteed bound method, running on a MasPar MP-1 with 4096 processing elements and 64 kilobytes of memory for each PE:

Problem Number 1 -- Problem Dimensions: m = 5 n = 10
 Random seed: -798398917

Matrix Function:

```
-0.586 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000
+0.000 -0.768 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000
+0.000 +0.000 -0.864 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000
+0.000 +0.000 +0.000 -0.593 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000
+0.000 +0.000 +0.000 +0.000 -0.248 +0.000 +0.000 +0.000 +0.000 +0.000
+0.000 +0.000 +0.000 +0.000 +0.000 -0.840 +0.000 +0.000 +0.000 +0.000
+0.000 +0.000 +0.000 +0.000 +0.000 +0.000 -0.173 +0.000 +0.000 +0.000
+0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 -0.565 +0.000 +0.000
+0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 -0.321 +0.000
+0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 +0.000 -0.813
```

Vector h:

```
-19.76 +14.38 -47.52 +24.105 +42.18 +39.88 +17.77 +38.806 +6.196 +12.716
```

Matrix Starting A:

```
+0.713 -7.564 -13.232 +8.429 +6.411 -4.320 -4.301 -7.525 +12.610 +10.241
-19.698 +4.986 +6.242 +15.145 -5.845 +4.689 +0.979 +5.385 +1.951 +9.952
-4.860 +4.377 +4.907 +13.085 +3.240 +5.408 +6.036 -7.723 -6.701 -1.441
+5.418 -3.565 +24.156 +13.492 -3.017 -14.292 +0.688 -2.044 +17.23 +0.189
```

+1.000 +1.000 +1.000 +1.000 +1.000 +1.000 +1.000 +1.000 +1.000 +1.000

Vector Starting b:

+118.160 +733.286 +309.216 +1160.593 +373.332

bounds[0]: lower: 0.000000000000000000 upper: 329.59983721840291082
 bounds[1]: lower: 0.000000000000000000 upper: 252.75191414283514745
 bounds[2]: lower: 0.000000000000000000 upper: 115.31460070840883247
 bounds[3]: lower: 0.000000000000000000 upper: 113.09135520516593942
 bounds[4]: lower: 0.000000000000000000 upper: 231.34304683806431057
 bounds[5]: lower: 0.000000000000000000 upper: 233.43878529537877852
 bounds[6]: lower: 0.000000000000000000 upper: 229.83744880397534871
 bounds[7]: lower: 0.000000000000000000 upper: 322.42173995029997968
 bounds[8]: lower: 0.000000000000000000 upper: 127.36488482568393010
 bounds[9]: lower: 0.000000000000000000 upper: 143.91894621580291868
 recursion level #1 entered

Changing upper bound 5 to 116.71939264768938926409

Changing upper bound 9 to 71.95947310790145934334
 recursion level #2 entered

Changing lower bound 0 to 164.79991860920145541058

Changing upper bound 1 to 63.18797853570878686469

Changing upper bound 2 to 57.65730035420441623728

Changing upper bound 3 to 56.54567760258296971188

Changing upper bound 4 to 115.67152341903215528873

Changing upper bound 5 to 58.35969632384469463204

Changing upper bound 6 to 114.91872440198767435504

Changing lower bound 0 to 247.19987791380219732673

Changing upper bound 1 to 31.59398926785439343234

Changing upper bound 2 to 28.82865017710220811864

Changing upper bound 3 to 28.27283880129148485594

Changing upper bound 4 to 57.83576170951607764437

Changing lower bound 5 to 29.17984816192234731602

Changing upper bound 6 to 57.45936220099383717752

Changing upper bound 7 to 161.21086997514998984116

Changing upper bound 8 to 63.68244241284196505148

Changing upper bound 9 to 35.97973655395072967167
recursion level #2 entered

-- Global Minimum Vertex (non-zero components only)--

x[0] = 329.600

x[5] = 43.732

Global Min = -37378.800

Lower Bound = -37379.028

Number of Incumbent Improvements = 3

Number of Subregion Eliminations = 19

Elapsed CPU Time = 24.670 Seconds

Random Number Seed Value = -798398917

Domain Splitting Occurred 1 Times

Epsilon Tolerance or Pruning Satisfied 2 Times

Total Number of LP Problems = 126

Appendix B: Subregion Elimination

The current method of subregion elimination proceeds by first constructing underestimators for each half of the hyperrectangle along each concave dimension (see Chapter III, Section 3, on page 37). This method can throw away at most half of the feasible region in each of the concave dimensions, and can in fact result in no eliminations at all in any specific dimension. The question arises as to how one might pick a point, different from the midpoint, at which to construct the underestimators so that the *maximum* possible subsection of the feasible region is thrown away for each concave dimension. For example, if minimizing the first underestimator in a specific dimension over the first "two-thirds" of the hyperrectangle allows that entire region to be discarded, it would be highly desirable to have some method of knowing a priori that the "two-thirds point" was the proper place at which to construct the underestimators. This problem can be formulated for the first underestimator in each concave dimension as

$$\begin{aligned} &\text{maximize } x_i \\ &\text{such that} \\ &\bar{\varphi}_i^{(1)}(\bar{x}_i^{(1)}) + \bar{\varphi}_{i'}(\bar{x}_{i'}^{(1)}) > \varphi(\tilde{\mathbf{x}}), \end{aligned}$$

where $\bar{\mathbf{x}}^{(1)}$ is the optimal solution over the feasible region to the problem $\min \bar{\varphi}^{(1)}(\mathbf{x})$, $\bar{\varphi}_{i'}(\bar{x}_{i'}^{(1)}) = \bar{\varphi}(\bar{\mathbf{x}}^{(1)}) - \bar{\varphi}_i(\bar{x}_i^{(1)})$, and $\bar{\varphi}_i^{(1)}(\bar{x}_i^{(1)}) = \gamma_i^{(1)} + c_i^{(1)} \bar{x}_i^{(1)}$. In addition, this last equation can be broken down into its constituents as $c_i^{(1)} = (\varphi_i(x_i) - \varphi_i(\alpha_i)) / (x_i - \alpha_i)$ and $\gamma_i^{(1)} = \varphi_i(\alpha_i) - c_i^{(1)} \alpha_i$.

This leads to the relation

$$\varphi_i(\alpha_i) + \frac{(\varphi_i(x_i) - \varphi_i(\alpha_i))}{(x_i - \alpha_i)} \left(\bar{x}_i^{(1)} - \alpha_i \right) + \bar{\varphi}(\bar{\mathbf{x}}^{(1)}) - \bar{\varphi}_i(\bar{x}_i^{(1)}) > \varphi(\tilde{\mathbf{x}}).$$

Multiplying through by $x_i - \alpha_i$ and rearranging the terms in order of complexity results in the following restatement of the original problem:

maximize x_i

such that

$$\begin{aligned} & \varphi_i(x_i) \left(\bar{x}_i^{(1)} - \alpha_i \right) + \left(\bar{\varphi}(\bar{\mathbf{x}}^{(1)}) - \bar{\varphi}_i(\bar{x}_i^{(1)}) \right) (x_i - \alpha_i) \\ & + (\varphi_i(\alpha_i) - \varphi(\tilde{\mathbf{x}})) (x_i - \alpha_i) - \varphi_i(\alpha_i) \left(\bar{x}_i^{(1)} - \alpha_i \right) > 0 \end{aligned}$$

where $\bar{\mathbf{x}}^{(1)}$ is the optimal solution over the feasible region to the problem $\min \bar{\varphi}^{(1)}(\mathbf{x})$.

In this equation, the second term is bilinear in $\bar{\mathbf{x}}^{(1)}$ and x_i , the third term is linear in x_i , and the last term is linear in $\bar{\mathbf{x}}^{(1)}$. However, the first term makes this equation quite complex, as it is a combination of a linear function of $\bar{x}_i^{(1)}$ and a concave function of x_i . Since this is an optimization problem with only a single constraint, it might be possible to solve this so that more of the feasible region could be eliminated. However, $\bar{\mathbf{x}}^{(1)}$ is also a variable, and it is dependent in a nonlinear manner on the initial choice for x_i , so solving this problem of optimizing the subregion elimination step is possibly as hard as solving the original problem.

In conclusion, while there are many possible methods of attempting to throw away larger parts of the feasible region, it is quite difficult to optimize this step, so a simple algorithm which constructs the underestimators at the midpoint of the hyperrectangle appears to be a quick and efficient method of implementation.

References

1. AMDAHL, G., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Conference Proceedings* **30**, 483-85, Thompson Books, Washington, D.C., Apr. 1967.
2. BENDSOE, M.P., A. BEN-TAL, and J. ZOWE, "Optimization Methods for Truss Geometry and Topology Design," *Structured Optimization* **7**, 141-159 (1994).
3. BOENDER, C.G.E., and A.H.G. RINNOOY KAN, "Bayesian Stopping Rules for Global Optimization Methods," *Mathematical Programming* **37**, 59-80 (1987).
4. BROWN, G.G., and S. LAWPHONPHANICH, "Optimizing Ship Berthing," *Naval Research Logistics* **41**, 1-15 (1994).
5. BYRD, R.H., et al, "Concurrent Stochastic Methods for Global Optimization," *Mathematical Programming* **46**, 1-29 (1990).
6. FLOUDAS, C.A., and P.M. PARDALOS, "A Collection of Test Problems for Constrained Global Optimization Algorithms," *Lecture Notes in Computer Science* **455**, Springer Verlag, 1990.
7. GEIST, A., et al, *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, Oak Ridge, Tenn., Sep. 1994.
8. GLINSMAN, D.H., and J.B. ROSEN, "Constrained Concave Quadratic Global Minimization by Integer Programming," *Technical Report TR 86-37*, University of Minnesota Department of Computer Science, Minneapolis, Minn., 1986.
9. KARATZAS, G.P., and G.F. PINDER, "Groundwater Quality Management Using a 3-D Numerical Simulator and a Cutting Plane Optimization Method," *Computational Methods in Water Resources* **X**, 841-848 (1994).
10. LUENBERGER, D.G., *Linear and Nonlinear Programming*, Addison-Wesley Publishing Company, Reading, Mass., 1984.
11. MARKOWITZ, H.M., *Mean-Variance Analysis in Portfolio Choice and Capital Markets*, Blackwell, New York, 1987.
12. McLAUGHLIN, M.F., "Implementation of a Parallel Stochastic Solving Method for Linearly Constrained Concave Global Optimization Problems Using Parallel Computing," *Trident Scholar Project Report 190*, United States Naval Academy,

Annapolis, Md., 1992.

13. PARDALOS, P.M., "Integer and Separable Programming Techniques for Large-Scale Global Optimization Problems," *Ph.D. Thesis*, University of Minnesota Computer Science Department, Minneapolis, Minn., 1985.
14. PARDALOS, P.M., and J.B. ROSEN, "Constrained Global Optimization: Algorithms and Applications," *Lecture Notes in Computer Science* **268**, Springer Verlag, 1987.
15. PARDALOS, P.M., and G. SCHNITGER, "Checking Local Optimality in Constrained Quadratic Programming is NP-Hard," *Operations Research Letters* **7**, No. 1, 33-35 (1988).
16. PHILLIPS, A.T., "Parallel Algorithms for Constrained Optimization," *Ph.D. Thesis*, University of Minnesota Department of Computer Science, Minneapolis, Minn., 1988.
17. PHILLIPS, A.T., and J.B. ROSEN, "A Parallel Algorithm for Constrained Concave Quadratic Global Minimization: Computational Aspects," *Technical Report TR 87-59*, University of Minnesota Institute of Technology, Minneapolis, Minn., Dec. 1987.
18. QUINN, M.J., *Parallel Computing: Theory and Practice*, 2nd ed., McGraw-Hill, New York, N.Y., 1994.
19. ROCKAFELLAR, R.T., *Convex Analysis*, Princeton University Press, Princeton, N.J., 1970.
20. ROSEN, J.B., "The Gradient Projection Method for Nonlinear Programming, Part I. Linear Constraints," *J. SIAM* **8**, 181-217 (1960).
21. ROSEN, J.B., "The Gradient Projection Method for Nonlinear Programming, Part II. Nonlinear Constraints," *J. SIAM* **9**, 514-532 (1961).
22. SHECTMAN, J.P., and N.V. SAHINIDIS, *A Finite Algorithm for Global Minimization of Separable Concave Programs*, University of Illinois, Urbana, Ill., Jan. 1995.
23. ZILVERBERG, N.D., "Global Minimization for Large Scale Linear Constrained Systems," *Ph.D. Thesis*, University of Minnesota Department of Computer Science, Minneapolis, Minn., 1983.